Staged Grid NewSQL Database System for OLTP and Big Data Applications

by

Lengdong Wu

A thesis submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Department of Computing Science
University of Alberta

Abstract

Big data applications demand and consequently lead to developments of diverse scalable data management systems, ranging from NoSQL systems to the emerging NewSQL systems. In order to serve thousands of applications and their huge amounts of data, data management systems must be capable of scale-out to clusters of commodity servers. The overarching goal of this dissertation is to propose principles, paradigms and protocols to architect efficient, scalable and practical NewSQL database systems that address the unique set of challenges posed by the big data trend. This dissertation shows that with careful choice of design and features, it is possible to implement scalable NewSQL database systems that efficiently support transactional semantics to ease application design. In this dissertation, we first investigate, analyze and characterize current scalable data management systems in depth and develop comprehensive taxonomies for various critical aspects covering the data model, the system architecture and the consistency model. On the basis of analyzing the scalability limitations of current systems, we then highlight the key principles for designing and implementing scalable NewSQL database systems.

This dissertation advances the state-of-the-art by improving and providing satisfactory solutions to critical facets of NewSQL database systems. In particular, first we specify a staged grid architecture to support scalable and efficient transaction processing using clusters of commodity servers. The key insight is to disintegrate and reassemble system components into encapsulated staged modules. Effective behavior rules for communication are then defined to orchestrate independent staged modules deployed on networked computing nodes into one integrated system. Second, we propose a new formula-based protocol for distributed concurrency control to support thousands of concurrent users accessing data distributed over commodity servers. The formula protocol for concurrency is a variation of the multi-version time-stamp concurrency control protocol, which guarantees serializability. We reduce the overhead of conventional implementation by technologies including logical formula caching and dynamic timestamp ordering. Third, we identify a new consistency model-BASIC (Basic Availability, Scalability, Instant Consistency) that matches the requirements where extra efforts are not needed to manipulate inconsistent soft states of weak consistency models. BASIC extends the current understanding of CAP theorem by characterizing precisely different degree of dimensions that can be achieved rather than simply what cannot be done.

We introduce all these novel ideas and features based on the implementation of Rubato DB, a highly scalable NewSQL database system. We have conducted extensive experiments that clearly show that Rubato DB is highly scalable with efficient performance under both TPC-C and YCSB benchmarks. These results verify that the staged grid architecture and the formula protocol provide a satisfactory solution to one of the important challenges in the NewSQL database systems: to develop a highly scalable database management system that supports various consistency levels from ACID to BASE.

Preface

The research conducted for this thesis forms part of a research collaboration, led by Professor Li-Yan Yuan and Professor Jia-Huai You at the University of Alberta. Most of the research described in this thesis has been previously published. The following paragraph lists the original publications and describes my contribution to each of them.

Chapter 2 is based on a publication in JCST [Wu et al., 2015]. In this paper, I developed the taxonomies and principles with the assistance of Professor Li-Yan Yuan. Chapters 3, 4, 5 are based on the publications for the system Rubato DB appeared in CIKM [Yuan et al., 2014] and SIGMOD [Yuan et al., 2015]. I was responsible for some components implementation, architecture and algorithm verification, all the experiments and analysis as well as the manuscript composition. Chapter 6 is based on a publication that appeared in IEEE Big Data [Wu et al., 2014]. In this paper, I proposed the model, designed the protocol, conducted the analysis and implemented the experiments.

Acknowledgements

I would like to express my deepest gratitude to my doctoral advisors Professor Dr. Li-Yan Yuan and Professor Dr. Jia-Huai You for their long-time support, caring and patience. Especially, I want to thank Professor Dr. Li-Yan Yuan for his excellent guidance and providing me with an excellent platform for doing research. Thanks for letting me experience the research in the big data trend, patiently leading me to grow as a researcher, and financially supporting my research continuously. What they have taught me is the most precious treasure of my life.

I would also like to thank my committee members, Professor Dr. Ke Wang, Professor Dr. Denilson Barbosa, Professor Dr. Davood Rafiei, for their brilliant comments and suggestions and making my defense an enjoyable moment. Many thanks to IT support staff in our department and Cybera, I could not finish the experiments without their hardware support.

A special thank goes to my family. Words cannot express how grateful I am to my parents, for all of the sacrifices that they've made on my behalf. Thank all of them to support me to strive towards my goal with their best wishes.

At the end I would like to express my most special appreciation to my beloved wife Huiling Zhou, who is always there cheering me up and stands by me. Thank her for being always my support in every moment.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

"The beginning is the most important part of the work."

- Plato.

1.1 Development of Large-Scale Data Management Systems

Data is flowing into organizations, ranging from traditional database vendors to new emerging internet enterprises to social research institutes, at an unprecedented scale. Data volumes collected by many companies are doubled in less than a year or even sooner. The amount of data is growing with a speed faster than the "Moore's Law", which predicts that general-purpose hardware and software solutions that advance at the rate of Moore's Law will not be able to keep pace with the exploding data scale [Agrawal *et al.*, 2011]. In the database community, the "Big Data" problems arose when the traditional enterprise database products were not able to handle the tremendous business data and Web-scale data [Borkar *et al.*, 2012]. The pursuit for tackling the challenges posed by the big data management has given rise to a plethora of systems that possess their own desiderata revealing various possibilities in the design space.

To date, the trend of "big data" is usually characterized by the following dimensions according to the well-known cliché: *volume*, *velocity*, *variety* and *veracity*.

- *Volume*. Excessive data volumes and a large number of concurrent users require substantially throughput raising for the systems.

- *Velocity.* Data is flowing in at an unprecedented speed and needs to be dealt with in a timely manner.
- *Variety.* Data comes in all types of formats, from the structured relational data to the unstructured data.
- *Veracity.* Inconsistency or uncertainty of data, due to the quality of the data source or transmission latency, will jeopardize the utility and integrity of the data.

The "Big Data" trend has imposed challenges on the conventional design and implementation of traditional data management systems. In particular, the ability to scale out for processing an enhanced workload has become an important factor for the proliferation and popularization of data management systems. Big data needs to be partitioned across a large number of distributed servers, since no single server is capable of containing an entire copy of the big data. To design and implement such a large-scale system, multiple, sometimes competing goals require to be achieved. In particularly, the desirable properties include:

- *High scalability.* The system scales out linearly by adding more computing resources to achieve enhanced throughput and storage capacity.
- *High Availability.* All continuously operations issued to the system response successfully. No operation will block indefinitely or return an error indicating unavailable data.
- *Low latency.* Operations complete in a timely manner. Commercial service-level objectives suggest average performance of a few milliseconds and worse-case performance (i.e., 99.9th percentile) of 10s or 100s of milliseconds [DeCandia *et al.*, 2007].
- *Partition Tolerance.* In the deployment with hundreds of thousands of servers, the network partitions will become rather common than rare. The system is capable of continuing to operate under network partition.

- *Strong Consistency.* An ideal system would provide strong consistency, which dictates that operations appear to take effect across the entire system at a single instance in time between the invocation and completion of the operation. A strong consistency simplifies the users experience by providing single images of objects across wide-area storage.

The pursuit for these challenges has given rise to a plethora of data management systems. Diverse systems explore various possibilities in the design spaces.

When considering the spectrum of historical data management system developments, we firstly have traditional relational database systems (RDBMSs) that provide low latency and high throughput of transaction processing, but lack the capacity of scale-out. As expected, conventional database vendors have recently developed their own system appliances in response to the high scalability requirement. Typically, Oracle Exadata[1], IBM Netezza[2] and Teradata[3] exploit the declarative nature of relational query languages and deliver high performance by leveraging a massively parallel fashion within a collection of storage cells. Oracle Exalytics [Gligor and Teodoru, 2011] is the industrial pioneer to use terabytes of DRAM, distributed across multiple processors, with a high-speed processor interconnect architecture that is designed to provide a single hop access to all the memory. Azure [Campbell *et al.*, 2010] is a parallel run-time system, utilizing specific cluster control with minimal invasion into the SQL Server code base. Some research prototypes, such as H-store [Kallman *et al.*, 2008], later commercialized into VoltDB[4], and C-Store [Stonebraker *et al.*, 2005], the predecessor of Vertica[5], also provide their tentative solutions for the big data management challenges. Though these systems have been proven to scale well, but there are very few known deployments with thousands of nodes to achieve ultimate scalability [Abouzeid *et al.*, 2009; Isard *et al.*, 2007; Shute *et al.*, 2013; Melnik *et al.*, 2010].

With the growing popularity of the Internet, many applications were delivered over

[1]Technique overview of the Oracle Exadata Database Machine and Exadata Storage Server. Oracle White Paper, 2012.

[2]The Netezza Data Appliance Architecture: A Platform for High Performance Data Warehousing and Analytics. IBM Red Paper, 2011.

[3]Teradata past, present, and future. `http://isg.ics.uci.edu/scalable_dml_lectures2009-10.html`

[4]https://voltdb.com/

[5]http://www.vertica.com/

the Internet and the scale of these applications also increased rapidly. As a result, many Internet companies, such as Google, Yahoo!, and Amazon, faced the challenge of serving hundreds of thousands to millions of concurrent users. Classical RDBMS technologies could not scale to these workloads while using commodity hardware to be cost-effective. In the absence of appropriate scalability of traditional relational database management systems, a notable phenomena - the NoSQL movement, began in early 2009 and is now growing rapidly. The need for highly available and scalable distributed data management systems with reliable and "always-writable" properties, leads to the development of Amazon Dynamo [DeCandia *et al.*, 2007] and Yahoo! PNUTS [Cooper *et al.*, 2008]. An open-source clone of Dynamo, Cassandra [Lakshman and Malik, 2010], has also been developed by the Apache community. Oracle NoSQL Database [Joshi *et al.*, 2012] using Oracle Berkeley DB as the underlying data storage engine provides flexible durability and consistency polices. Similar systems such as Voldemort[6] and SimpleDB[7], are all categorized as key-value data stores. Key-value data stores are characterized as simplified highly scalable databases addressing properties of being schema-free, simple-API, horizontal scalability and relaxed consistency.

Google responds to the web-scale storage challenges by developing a family of systems. Google File System (GFS) [Ghemawat *et al.*, 2003] is a distributed file system for large distributed data-intensive applications, providing with an OS-level byte stream abstraction on a large collection of commodity hardware. The Bigtable [Chang *et al.*, 2008] is a hybrid data storage model built on GFS. Megastore [Baker *et al.*, 2011] and Spanner [Corbett *et al.*, 2012] are two systems over the Bigtable layer. Megastore blends the scalability and the fault tolerance ability of Bigtable with transactional semantics over distant data partitions. Spanner is a multi-version, globally-distributed and synchronously replicated database by adopting "True Time" that combines an atomic clock with a GPS clock for synchronization across world-wide datacenters. HBase[8] and Hypertable[9] provide open-source versions of Google's Bigtable.

[6]http://project-voldemort.com/
[7]http://aws.amazon.com/en/simpledb/
[8]http://hbase.apache.org/
[9]http://hypertable.org/

Google developed the MapReduce framework that is highly scalable and parallel for big data processing [Dean and Ghemawat, 2008]. Taking the released MapReduce paper as the guideline, open-source equivalents were developed as well, such as the Apache Hadoop MapReduce platform built on the Hadoop Distributed File System (HDFS) [Shvachko *et al.*, 2010]. Numerous NoSQL systems based on the MapReduce and Hadoop utilize a large collection of commodity servers to provide high scalability. For example, a set of systems with high-level declarative languages, including Yahoo! Pig [Olston *et al.*, 2008], Microsoft SCOPE [Chaiken *et al.*, 2008] and Facebook Hive [Thusoo *et al.*, 2009], are realized by compiling queries into the MapReduce framework before the execution on the Hadoop platform. Greenplum [Cohen *et al.*, 2009] integrates the ability to write MapReduce functions over data stored in their parallel database products. HadoopDB [Abouzeid *et al.*, 2009] is to connect multiple single-node database systems using Hadoop as the task coordinator and the network communication layer. Queries are parallelized across nodes using the MapReduce framework. There are also a number of commercialized systems that combine parallel DBMSs with the MapReduce framework such as Cloudera[10], Teradata Aster[11], etc.

1.2 Motivation and Challenges

NoSQL systems, typically like key-value stores, were designed to scale to thousands of commodity servers, and to replicate data across geographically remote locations, while supporting low latency and highly available accesses. NoSQL systems support simple functionality based on single-key operations, while being lack of transactional guarantees for accesses spanning multiple keys. However, as the class of applications broadened, applications often access multiple data items within a single request. These applications range from online social applications to enterprise class applications deployed in the scalable grid environment. Providing transactional support in the grid has therefore been an active area of research.

Classical transactions guarantee the ACID semantics: either all operations in a transaction execute or none of them execute (Atomicity), a transaction takes the database from one consistent state to another consistent state (Consistency), concur-

[10]https://www.cloudera.com
[11]http://www.asterdata.com/

rently executing transactions do not interfere (Isolation), and updates made from a committed transaction are persistent (Durability) (ACID). Serializability, the strongest form of isolation, ensures that transactions execute in an order equivalent to serial order, thus making concurrency transparent to the application developers. The ACID semantics, therefore, considerably simplifies application logic and helps reasoning about correctness and data consistency using sequential execution semantics.

NoSQL systems represent the recent evolution in building infrastructure by making trade-off between scalability and consistency, which often renounce strict consistency (e.g. ACID) and turn to weak consistency (e.g. BASE [Vogels, 2008]) for high scalability. Such single key accesses in NoSQL systems were enough to support the class of applications that the key-value stores initially targeted. However, as the class of applications using the NoSQL systems diversified, applications accessing multiple data items within a single request was inevitable. NoSQL systems do not provide any atomicity and isolation guarantees for such accesses to multiple data items. In the absence of transactional guarantees, applications must either embrace the data inconsistencies, which considerably complicates reasoning about data freshness and application correctness.

As a result, there exists a big gap between RDBMSs that provide strong transactional guarantees but are hard to scale-out and NoSQL systems that scale-out without supporting transactional semantics guarantees. The recently proposed NewSQL database systems (relative to NoSQL) aim to achieve the scalability and availability same as NoSQL systems while preserving the ACID properties for transactions and complex functionality of relational databases [Stonebraker, 2012]. NewSQL database systems are appropriate in applications where traditional RDBMS have been used, but requiring additional scalability and performance enhancement. With the ability to ingest large amounts of data quickly and scale to meet growing demand, NewSQL database systems support rapid analytics and decision making, helping organizations leverage their data for business advantage. Particularly, NewsSQL database systems are notable to support enterprise initiatives such as fraud detection, digital advertising, market segmentation analysis, real-time pricing and billion, retail loyalty programs, as they provide the scalable performance of NoSQL systems for OLTP workloads as well as safety guarantees for critical data.

According to the core requirement differences between NoSQL systems and NewSQL systems, there are challenges in the design and implementation of NewSQL systems in the following aspects:

- NoSQL systems usually merely support non-relational data model such as key-value stores [DeCandia *et al.*, 2007] and Bigtable-like stores [Chang *et al.*, 2008]. NoSQL systems represent a recent evolution by making trade-off between scalability and complexity of the data model. Thus the implementation of NewSQL systems faces with the first issue:

  Is the complex relational data model an obstacle of scalability?

- NoSQL systems represent an evolution in building scalable infrastructure by sacrificing strong consistency and opting for weak consistency [Cooper *et al.*, 2008; DeCandia *et al.*, 2007; Lakshman and Malik, 2010]. However, strong consistency levels such as ACID properties are essential for NewSQL systems to manage critical data which requires safety guarantee. The implementation of NewSQL systems needs to resolve the second challenge:

  Is it possible to achieve high scalability with ACID?

- Traditional database vendors employ new techniques to explore the scalability of the symmetric multiple processing (SMP) architecture and the massively parallel processing (MPP) architecture. However, because of the inherent deficiencies due to the resources contention, the scalability of these architectures is not comparable with NoSQL systems. Some NoSQL systems based on the Bigtable [Chang *et al.*, 2008] and MapReduce framework [Dean and Ghemawat, 2008] utilize the shared-nothing infrastructure to provide high scalability. A set of systems with high-level declarative languages, including Yahoo! Pig [Olston *et al.*, 2008] and Facebook Hive [Thusoo *et al.*, 2009], are realized to compile queries into the MapReduce framework on the Hadoop platform.

  To achieve high performance and scalability, innovative software architecture should be applied to NewSQL systems, thus we need to consider:

Is it possible to scale out commonly used single server database system design?

1.3 Dissertation Overview

The overarching goal of this dissertation is to propose principles, protocols and paradigms to architect efficient, scalable and practical NewSQL database systems that address the unique set of challenges encountered by traditional DBMS in the big data trend. This dissertation shows that with careful choice of design and features, it is possible to implement scalable NewSQL database management systems that efficiently support transactional semantics to ease application design.

In this dissertation, we first investigate, analyze and characterize current scalable systems in depth and develop comprehensive taxonomies for various critical aspects covering the data model, the architecture, and the consistency model. On the basis of analyzing the scalability limitations of current systems, we then highlight the key principles for designing and implementing NewSQL database systems. Using these principles as the cornerstone, this dissertation advances the state-of-the-art by improving and providing satisfactory solutions to critical facets of NewSQL database systems.

We specify a staged grid architecture and abstractions to support scalable and efficient transaction processing using clusters of commodity servers. The key insight is to disintegrate and reassemble system components into encapsulated staged modules. Effective behavior rules for communication are then defined to orchestrate independent staged modules deployed on networked computing nodes into one integrated system. We verify the determinism of the proposed staged grid architecture based on the formalisms of Kahn's PN model with enhancements that are designed to make it efficiently implementable and preserve the properties including scalability, composability and boundedness.

A new formula-based protocol is proposed for distributed concurrency control to support for thousands of concurrent users accessing data distributed over commodity servers. The formula protocol for concurrency is a variation of the multi-version timestamp concurrency control protocol, which guarantees the serializability. We reduce the overhead of conventional implementation by technologies including logical formula caching and dynamic timestamp ordering.

A novel consistency model-BASIC (Basic Availability, Scalability, Instant Consistency) is identified that matches the requirements where extra efforts are not needed to manipulate inconsistent soft states of weak consistency models. BASIC extends the current understanding of CAP theorem by characterizing precisely different degree of dimensions that can be achieved rather than simply what cannot be done.

We introduce all these ideas and features based on the implementation of Rubato DB, a highly scalable NewSQL database system. We have conducted extensive experiments that clearly show that Rubato DB is highly scalable with efficient performance under both TPC-C and YCSB benchmarks. These results verify that the staged grid architecture and the formula protocol provide a satisfactory solution to one of the important challenges in the NewSQL database systems: to develop a highly scalable database management system that supports various consistency levels from ACID to BASE.

1.4 Contributions and Impact

This dissertation makes several fundamental contributions towards realizing our vision of building scalable NewSQL database systems. Our contributions significantly advance the state-of-the-art by supporting scale-out architecture and concurrency control for transaction processing in the NewSQL database systems. These technologies are critical to ensure the success of the next generation of NewSQL database systems serving the big data trend. We now highlight these contributions and their impact.

- We categorize and analyze critical aspects of NewSQL database systems, and identify the principles for designing and developing NewSQL database systems. We propose taxonomies to classify techniques based on multiple dimensions, in which every high scalable data management system is able to find its position. A thorough understanding of current systems and a precise classification are essential for analyzing the scalability limitations.

- We define a highly scalable staged grid database architecture, and provide its implementation techniques in details, based on which a NewSQL database system Rubato DB is implemented. We disintegrate and reassemble the traditional single server database system design into multiple encapsulated staged modules. Then independent staged modules deployed on networked computing nodes are orchestrated based on communication to work as one single database instance. We verify the determinism property of the staged grid architecture.

- We propose the formula protocol for concurrency in the distributed environment ensuring serializability for transation processing. We describe the novel technologies employed for the implementation of the formula protocol (i.e., logical transformation formula caching and dynamic timestamp ordering) to achieve strong consistency with high scalability and availability.

- We explicitly identify important disadvantages of weak consistency (e.g. BASE) with tangible examples and formally define BASIC property, which is an alternatively higher level of consistency than BASE, while lower than ACID. BASIC is desired by a largely class of applications requiring multi-partition, partial-replication, atomically non-transactional operational data access where none of inconsistent states should be visible to clients. We generalizes the understanding of CAP theorem based on BASIC. Given the general extension of CAP theorem, we do not just pick two of the three properties, while the optimal three can be chosen based on different application requirements.

- We conducted extensive experiments focused on the performance of Rubato DB, which shows that it supports various consistency guarantees ranging from ACID to BASE with scalability. Our experiments not just confirm our solution but also provides much needed insights towards NewSQL database design and implementation.

1.5 Synopsis

This thesis is organized as follows. Chapter 2 propose taxonomies to classify techniques based on multiple dimensions, in which every high scalable system is able to find its position. A thorough understanding of current systems and a precise classification are provided for analyzing the scalability limitations. Chapter 3 discusses the overview design of a NewSQL database system-Rubato DB. Chapter 4 proposes a highly scalable staged grid database architecture, and provide its implementation techniques in details, based on which a NewSQL database system Rubato DB is implemented. Chapter 5 describes details on the formula protocol for concurrency in the distributed environment ensuring serializability for transaction processing. Chapter 6 identify important disadvantages of BASE and formally define BASIC property, which is an alternatively higher level of consistency than BASE, while lower than ACID. Chapter 7 concludes this dissertation.

Chapter 2

Taxonomy of Large-Scale Data Management System

"What is real is rational and what is rational is real."

- Hegel.

In this chapter, we provide a comprehensive study of the state-of-the-art large-scale data management systems and conduct an in-depth analysis on the critical aspects in the design of different infrastructures. We propose taxonomies to classify techniques based on multiple dimensions, in which every high scalable system is able to find its position. A thorough understanding of current systems and a precise classification are essential for analyzing the scalability limitations. Based on our taxonomies and analysis, we identify the principles for the implementation of large scale data management systems.

2.1 Data Model

Data model consists of two essential levels in the context of databases: the physical level and the conceptual level. The details of how data is stored in the database belong to the physical level[1]. The schema specifying the structure of the data stored in the database is described in the conceptual level.

[1]By physical level we mean a lower level of storage schema, not actual file structures on disk.

2.1.1 Physical Layout

A key factor affecting the performance for any data management system is the storage layout on the physical level used to organize the data on the storage disks. There are three mechanisms to map the two-dimensional (2D) relational tables onto the one-dimensional (1D) physical storage, i.e., row-oriented layout, column-oriented layout, and hybrid-oriented layout.

Row-Oriented Layout

Data has been organized within a block in a traditional row-by-row format, where all attributes data for a particular row is stored sequentially within a single database block. Traditional DBMSs towards ad-hoc querying of data tend to choose the row-oriented layout.

Column-Oriented Layout

Data is organized in a significant deviation of the row-oriented layout. Every column is stored separately in the column-oriented layout and values in a column are stored contiguously. Analytical applications, in which attribute-level access rather than tuple-level access is the frequent pattern, tend to adopt the column-oriented layout. They can then take advantage of the continuity of values in a column such that only necessary columns related with the queries are required to be loaded, reducing the I/O cost significantly [Abadi *et al.*, 2008].

Hybrid-Oriented Layout

The design space for the physical layout is not limited to merely row-oriented and column-oriented layouts, but rather that there is a spectrum between these two extremes, and it is possible to build the hybrid layout combining the advantages of purely row and column oriented layouts.

Hybrid-oriented layout schemes are designed based on different granularity. The most coarse-grained granularity essentially adopts different layouts on different replicas like fractured mirrors [Ramamurthy *et al.*, 2003]. The basic idea is straightforward: rather than two disks in a mirror being physically identical, they are logically identical in which one replica is stored in the row-oriented layout while the other one is in the column-oriented layout. Fractured mirror can be regarded as a new form of RAID-1, and the query optimizer decides which replica is the best choice for corresponding query execution.

The fine-grained hybrid schema [Grund *et al.*, 2010; Hankins and Patel, 2003] integrates row and column layouts in the granularity of individual tables. Some parts of the table are stored with the row-oriented layout, while other parts apply the column-oriented layout. An even finer schema is based on the granularity of disk blocks. Data in some blocks is aligned by rows while some is aligned by columns. To some extend, we can consider that, row-oriented layout, column-oriented layout are special extreme cases of hybrid-oriented layout.

2.1.2 Conceptual Model

Obtaining maximum performance requires a close integration between the physical layout and the conceptual schema. Based on the interpretation of data, three different conceptual data structures can be defined, i.e., unstructured data store, semi-structured data store and structured data store.

Unstructured data store

The unstructured data store is at the lowest conceptual level. All data items are uninterrupted, isolated and stored in a binary object or a plain file with no structure information. This type of data store takes the simplest data model: a map allowing requests to put and retrieve values per key. Operations are limited to single key/value pair without allowing cross-references between distinct pairs and there is no support for relational schema. With these restrictions and simplified query primitives, the key-value paradigm favors high scalability and performance advantages and provides developers the maximum flexibility to program customer features by their own. Nowadays, it gains popularity in large-scale web services [DeCandia *et al.*, 2007; Ghemawat *et al.*, 2003; Cooper *et al.*, 2008]. Due to the lack of structure information to extract data items separately, the row-oriented physical layout is the only choice for the unstructured data store.

Semi-structured data store

A semi-structured data store is used to store a collection of objects that is richer than the uninterrupted, isolated key/value pairs in the unstructured data store. A semi-structured data store, being schemaless, has certain inner structures known to applications and the database itself, and therefore can provide some simple query-by-value capability, but the application-based query logic may be complex [Chang *et al.*, 2008]. Because of its nature of schemaless, a semi-structured data store can only adopt row-oriented or hybrid layout on the physical layer.

Structured data store

A structured data store is used to store highly structured entities with strict relationships among them. Naturally, a structured data store is defined by its data schema, and usually supports comprehensive query facilities. As a representative of structured data store, the relational database organizes data into a set of tables, enforces a group of integrity constraints, and supports SQL as the query language.

2.1.3 Data Model Taxonomy

Based on the classification of the physical layout and the conceptual schema, we analyze currently prevailing database systems and categorize them in an appropriate taxonomy, as demonstrated in Figure 2.1, based on our observations outlined below.

Amazon's Dynamo [DeCandia *et al.*, 2007], Yahoo! PNUTS [Cooper *et al.*, 2008], Voldemort[2] and SimpleDB[3] are the typical systems belonging to the category with the row-oriented physical layout and the unstructured conceptual data store, since they are built on the simple key-value paradigm by storing data as binary objects identified by unique keys. These systems are all unstructured data stores that can only use the row-oriented physical layout.

Google's Bigtable [Chang *et al.*, 2008] together with Cassandra [Lakshman and Malik, 2010], HBase[4] and Hypertable[5], are representatives of semi-structured data store built on hybrid-oriented physical layout. They treat each individual table as a

[2]http://project-voldemort.com/
[3]http://aws.amazon.com/en/simpledb/
[4]http://hbase.apache.org/
[5]http://hypertable.org/

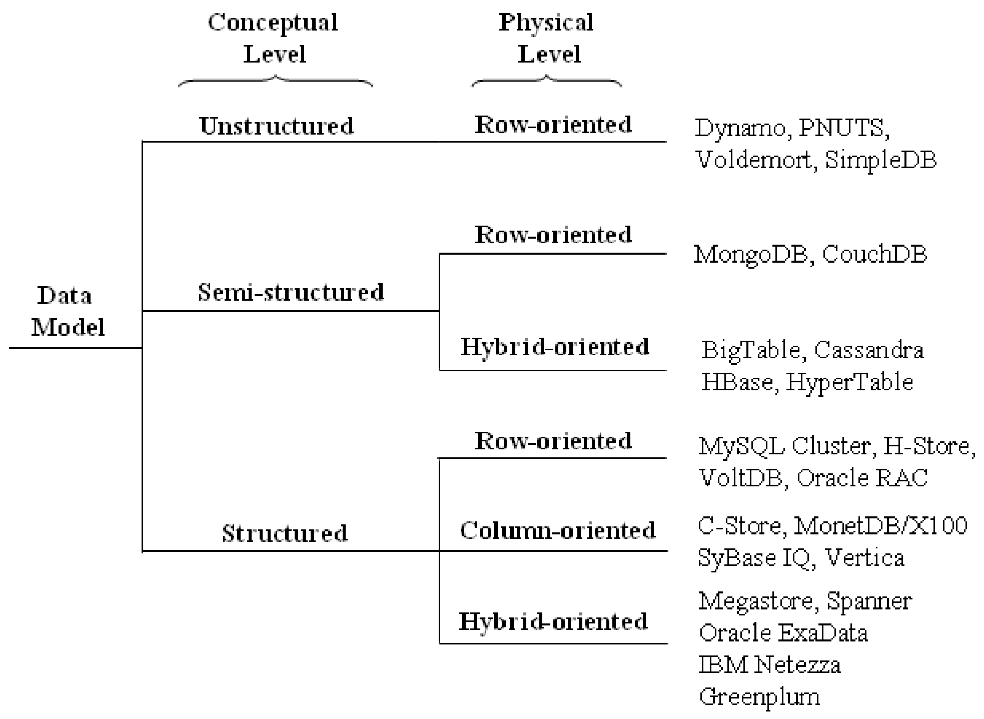

Conceptual Level
Physical Level
Data Model
Unstructured
Row-oriented
Dynamo, PNUTS, Voldemort, SimpleDB
Semi-structured
Row-oriented
MongoDB, CouchDB
Hybrid-oriented
BigTable, Cassandra HBase, HyperTable
Structured
Row-oriented
MySQL Cluster, H-Store, VoltDB, Oracle RAC
Column-oriented
C-Store, MonetDB/X100 SyBase IQ, Vertica
Hybrid-oriented
Megastore, Spanner Oracle ExaData IBM Netezza Greenplum

Figure 2.1: Taxonomy of Data Storage Model

sparse, distributed, multi-dimensional sorted map that provides the semi-structured data.

CouchDB[6] and MongoDB[7] considered as document stores[8], are another typical class of semi-structured data stores while using the row-oriented physical layout. Data in a document store is serialized from XML or JSON formats so that row-oriented layout is applied, similar to key-value stores.

C-store [Abadi *et al.*, 2006; Stonebraker *et al.*, 2005] supports the relational structured data model, whereas tables are stored column-oriented physically. MonetDB/X100 [Boncz *et al.*, 2006; Manegold *et al.*, 2009] and commercial systems Sybase IQ[9] and Vertica[10] adopt an idea similar to C-store. These systems benefit greatly from data caching and compressing techniques. Having data from each column with the same data type and low information entropy stored close together, the compression ratio can be dramatically enhanced to save a large amount of storage.

[6]http://couchdb.apache.org/
[7]http://www.mongodb.org/
[8]http://en.wikipedia.org/wiki/Document-oriented_database
[9]http://sybase.com/
[10]http://www.vertica.com/

Megastore [Baker *et al.*, 2011] and Spanner [Corbett *et al.*, 2012] define a structured data model based on relational tables stored on Bigtable [Chang *et al.*, 2008]. Since they are built on top of Bigtable, the hybrid layout is applied on the physical level. Same as traditional relational databases, the data model is declared in a schema. Tables are either entity group root tables or child tables, which must declare a single distinguished foreign key referencing a root table.

Oracle's Exadata[11], IBM Netezza[12] and Greenplum [Cohen *et al.*, 2009] evolved from traditional parallel database systems, and thus support the structured data store.

Furthermore, Exadata introduces Hybrid Columnar Compression (HCC) in the granularity of disk blocks, which employs the similar idea of Partition Attributes Across (PAX) [Ailamaki *et al.*, 2001] combined with compression. Netezza integrates row and column oriented layouts on each individual table. Greenplum provides multiple storage mechanisms with a variety of formats for different level of compression modes. The column-oriented store with the slightly compressed format is applied for data that is updated frequently, and append-only tables are using the row-oriented store with the heavily compressed format. These systems adopt the hybrid-oriented layout.

2.2 System Architecture

The system architecture is the set of specifications and techniques that dictate the way how various modules are orchestrated together within a system and how data processing logic works throughout the system.

In this section, we are going to classify systems according to diverse architectures. Basically, there are four important historical shifts in the architecture technology behind large-scale data management systems:

1. Invention of databases on a cluster of processors (single or multi-core) with shared memory;

[11] Technique overview of the Oracle Exadata Database Machine and Exadata Storage Server. Oracle White Paper, 2012.

[12] The Netezza Data Appliance Architecture: A Platform for High Performance Data Warehousing and Analytics. IBM Red Paper, 2011.

2. Improvement of databases on a cluster of processors with distributed memory but common storage disks;

3. Rise of parallel databases processing on the shared-nothing infrastructure;

4. Popularization of the MapReduce parallel framework and the distributed file system.

2.2.1 SMP on Shared-Memory Architecture

The symmetric multi-processing (SMP) on the shared-memory architecture, as illustrated in Figure 2.2, involves a pool of tightly coupled homogeneous processors running separate programs and working on different data with sharing common resources such as memory, I/O device, interrupt system and system bus. The single coherent memory pool is useful for sharing data and communication among tasks. This architecture is fairly common that most conventional database management systems have been deployed on such high-end SMP architectures.

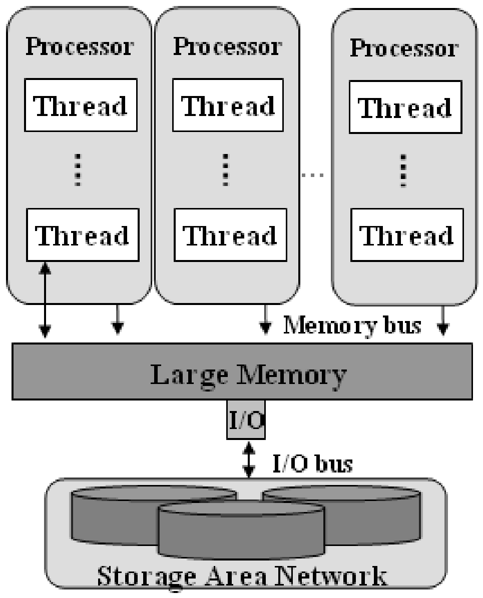


Figure 2.2: SMP on Shared-Memory Architecture

However, a small-scale SMP system consisting of a few processors is not capable of managing big data processing. It can be scaled "up" by adding additional processors, memories and disks devices, but is inevitably bounded by the resources limitation. In particular, when data volumes are increasing enormously, the memory bus bandwidth will be the ceiling for scaling-up, and similarly I/O bus bandwidth can also be clogged.

In addition, the initial expense of scaling up SMP server is quite high due to the larger capabilities and often more complex architectures [Poess and Nambiar, 2005]. It has been observed that the efficiency, scalability, and cost effectiveness of SMP systems degrade beyond 32 modern high performance microprocessors[13]. The SMP on shared-memory architecture has the disadvantage of limited scalability.

2.2.2 MPP on Shared-Disk Architecture

The massively parallel processing (MPP) on the shared-disk architecture is built on top of SMP clusters executing in parallel while sharing a common disk storage, as demonstrated in Figure 2.3. Each processor within an SMP cluster node shares the memory with its neighbors and accesses to the common storage across a shared I/O bus.

The shared-disk infrastructure necessitates disk arrays in the form of a storage area network (SAN) or a network-attached storage (NAS) [Gibson and Van Meter, 2000]. For instance, Oracle and HP grid solution orchestrates multiple small server nodes and storage subsystems into one virtual machine based on the SAN [Poess and Nambiar, 2005]. Unlike the shared-memory infrastructure, there is no common memory location to coordinate the sharing of the data. Hence explicit coordination protocols such as cache coherency [Bridge *et al.*, 1997] and cache fusion [Lahiri *et al.*, 2001] are needed [Poess and Nambiar, 2005].

The MPP on shared-disk architecture is commonly used in several well-known scalable database solutions. Two notable systems are Oracle's Exadata[14] and IBM Netezza[15]. Exadata is a complete, pre-configured Oracle system that combines Oracle RAC[16] with new Exadata Storage Servers. Exadata improves parallel I/O and filters only data of interest before transmitting. This process of filtering out extraneous data as early in the data stream as possible close to the data source can minimize

[13]Scaling-Up or Out. IBM Performance Technical Report. 2002.

[14]Technique overview of the Oracle Exadata Database Machine and Exadata Storage Server. Oracle White Paper, 2012.

[15]The Netezza Data Appliance Architecture: A Platform for High Performance Data Warehousing and Analytics. IBM Red Paper, 2011.

[16]http://www.oracle.com/technetwork/products/clustering/overview/index.html

the I/O bandwidth bottleneck and free up downstream components such as CPU and memory, thus having a significant multiplier effect on the performance. Netezza integrates server, storage and database all in a single compact platform. It proposes Asymmetric Massively Parallel Processing (AMPP) mechanism with Query Streaming technology that is an optimization on the hardware level .

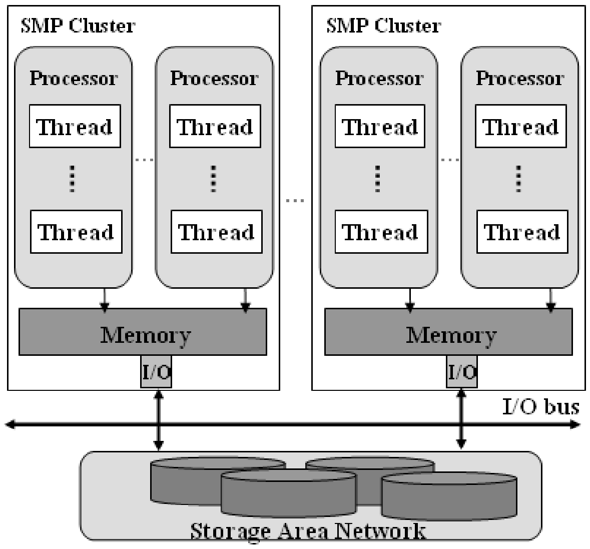


Figure 2.3: MPP on Shared-Disk Architecture

2.2.3 Sharding on Shared-nothing Architecture

Based on the idea that data management systems can be parallelized to leverage multiple commodity servers in a network to deliver increased scalability and performance, the parallelism on the shared-nothing infrastructure was coined for the new computing clusters. The sharding on the shared-nothing architecture is currently widely used in large-scale data management systems [Ghemawat *et al.*, 2003; Chang *et al.*, 2008; DeCandia *et al.*, 2007; Baker *et al.*, 2011; Cooper *et al.*, 2008; Lakshman and Malik, 2010; Campbell *et al.*, 2010].

In order to harness the power of this architecture, data is partitioned across multiple computation nodes. Common partitioning techniques used include range partitioning and hash partitioning. Range partitioning involves splitting the tables into non-overlapping ranges of their keys and then mapping the ranges to a set of nodes. In hash partitioning, the keys are hashed to the nodes serving them. These partitioning techniques are simple and are supported by most common database systems.

Each node hosts its own independent instance of the database system with operating on its portion of data. Each node is highly autonomous, performing its own scheduling, storage management, transaction management and replication. The autonomy allows additional nodes to be involved without concerning about interruption with others.

Sharding on shared-nothing architecture has a two-tier system design, as shown in Figure 2.4. The lower processing unit tier is composed of dozens to hundreds of processing machines operating in parallel. All query processing is decoupled at the processing unit tier. In the host tier, the assigned coordinator receives queries from clients and divides the query into a sequence of sub-queries that can be executed in parallel, and dispatches them to different processing units for execution. When processing units finish, the central host collects all intermediate results, handles post-processing and delivers results back to the clients. There are two flavors of this architecture that are **centralized topology** and **decentralized topology**.

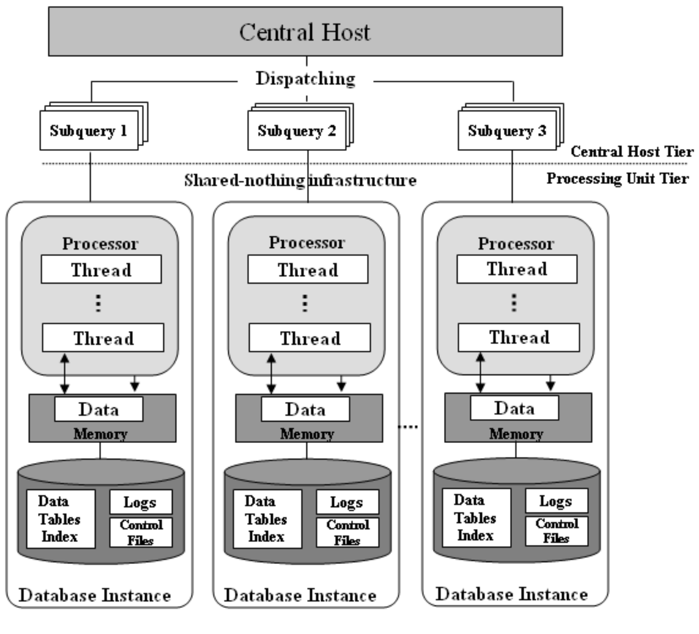


Figure 2.4: Dispatching on Shared-Nothing Architecture

Centralized Topology

Centralized topology utilizes a dedicated centralized coordinator to manage the system-wide membership state. The central server hosts the entire metadata and periodically communicates with each data server via heartbeat messages to collect the status of each member. The central server also takes charge of activities, typically including identifying the nodes that own the data with the key, routing the request to the nodes and integrating for the responses. The centralized topology simplifies the design and implementation of the complex architecture since the central node has an authoritative view of the whole system [Cooper *et al.*, 2008].

To prevent the central master server from easily becoming the bottleneck due to the heavy workload, shadow master mechanism is employed [Ghemawat *et al.*, 2003]. The key idea is to separate the control flow and the data flow of the system. The central master is only responsible for the metadata operation, while clients communicate directly with the data servers for reads and writes bypassing the central master. This design also delivers high aggregate throughput for high concurrent readers and writers performing a variety of tasks.

Decentralized Topology

Unlike the centralized topology, systems such as Dynamo [DeCandia *et al.*, 2007] and Cassandra [Lakshman and Malik, 2010] choose implementation of decentralized topology. All nodes take equal responsibility, and there are no distinguished nodes having special roles. This decentralized peer-to-peer topology excels the centralized one on the aspect of single point failure and workload balance. The gossip-based membership protocol [Birman, 2007] is a classical mechanism to ensure that every node keeps a routing table locally and is aware of the up-to-date state of other nodes. Consistent hashing [Karger *et al.*, 1997] is widely used in the decentralized topology implementation. Consistent hashing is a structure for looking up a server in a distributed system while being able to handle server failures with minimal effort. A client can send requests to any random node, and the node will forward the requests to the proper node along the ring.

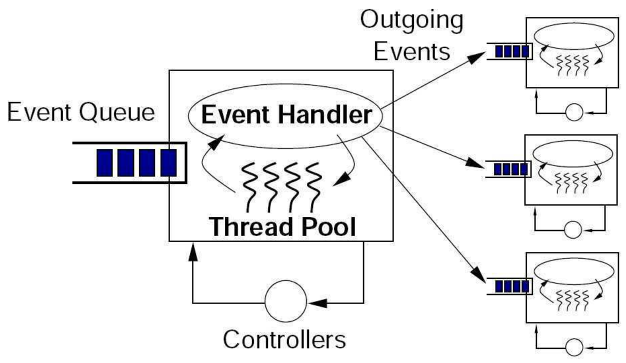


Figure 2.5: Staged Event Driven Architecture

2.2.4 MapReduce/Staged Event Driven Architecture

In the last decade, the importance of shared-nothing clusters was enhanced in the design of web services. Interesting architectures have been proposed to deal with massive concurrent requests on large data volumes for excessive user basis. One representative design is the well-known MapReduce framework for processing large data sets [Dean and Ghemawat, 2008]. Another design is the Staged Event-Driven Architecture (SEDA), which is intended to allow services to be well-conditioned for loading, preventing resources from being over-committed when the demand exceeds service capacity [Welsh *et al.*, 2001; Gribble, 2000].

Applications programmed with MapReduce framework are automatically parallelized and executed on a large cluster of commodity servers. The framework consists of two abstract functions, Map and Reduce, which can be considered as two different stages as well. The Map stage reads the input data and produces a collection of intermediate results; the following Reduce stage pulls the output from Map stage, and processes to final results. The trend of applying MapReduce framework to scale out configurations with lower-end, commodity servers has become popular, due to the drop in prices for the hardware and the improvement in performance and reliability.

Staged Event-Driven Architecture (SEDA) is designed based on the event-driven approach that has been introduced and studied for various software applications, such as Dynamic Internet Servers and high performance DBMSs [Harizopoulos and Ailamaki, 2003; Welsh *et al.*, 2001]. The event-driven approach implements the processing of individual task as a finite state machine (FSM), where transitions between states in the FSM are triggered by events. The basic idea of this architecture is that a software system is constructed as a network of staged modules connected with explicit queues, as illustrated in Figure 2.5 [Welsh *et al.*, 2001]. SEDA breaks the execution of applications into a series of stages connected by explicitly associated queues. Each stage represents a set of states from the FSM, and can be regarded as an independent, self-contained entity with its own incoming event queue. Stages pull a sequence of requests, one at a time, off their incoming task queue, invoke the application-supplied event handler to process requests, and dispatch outgoing tasks by pushing them into the incoming queue of the next stage. Each stage is isolated from one another for the purpose of easy resource management, and queues between stages decouple the execution of stages by introducing explicit control boundaries [Welsh *et al.*, 2001]. It has been shown that the aforementioned MapReduce framework can also be regarded as an architecture based on SEDA, and the basic MapReduce framework resembles the two-staged SEDA architecture. The general MapReduce extensions [Condie *et al.*, 2010; Verma *et al.*, 2013], introducing pipelined downstream data flow between multiple functional MapReduce pairs, behaves identically as SEDA [Isard *et al.*, 2007].

There has been some recent work on bringing ideas from MapReduce/SEDA to database management systems. The SEDA design has been applied to improve the staged database performance through exploiting and optimizing locality at all levels of the memory hierarchy of the single symmetric multiprocessing system at the hardware level [Harizopoulos and Ailamaki, 2003]. Some systems aim to integrate query construction into the MapReduce framework to allow greater data independence, automatic query optimization, and instruction reusability at the query level [Chaiken *et al.*, 2008; Olston *et al.*, 2008]. There are also attempts to program the MapReduce/SEDA over high performance parallel databases for a hybrid solution at the systems level [Cohen *et al.*, 2009; Abouzeid *et al.*, 2009; Thusoo *et al.*, 2009; Thusoo *et al.*, 2010].

2.2.5 System Architecture Taxonomy

Based on the above analysis, we present the taxonomy of Bigdata system architecture in Figure 2.6 [Wu *et al.*, 2015]. Due to the long-time popularity of the shared-memory multi-threads parallelism, almost all major traditional commercial DBMS providers support products with the SMP on shared-memory architecture, such as Microsoft SQL Server, Oracle Berkeley DB and Postgres-R, to name a few.

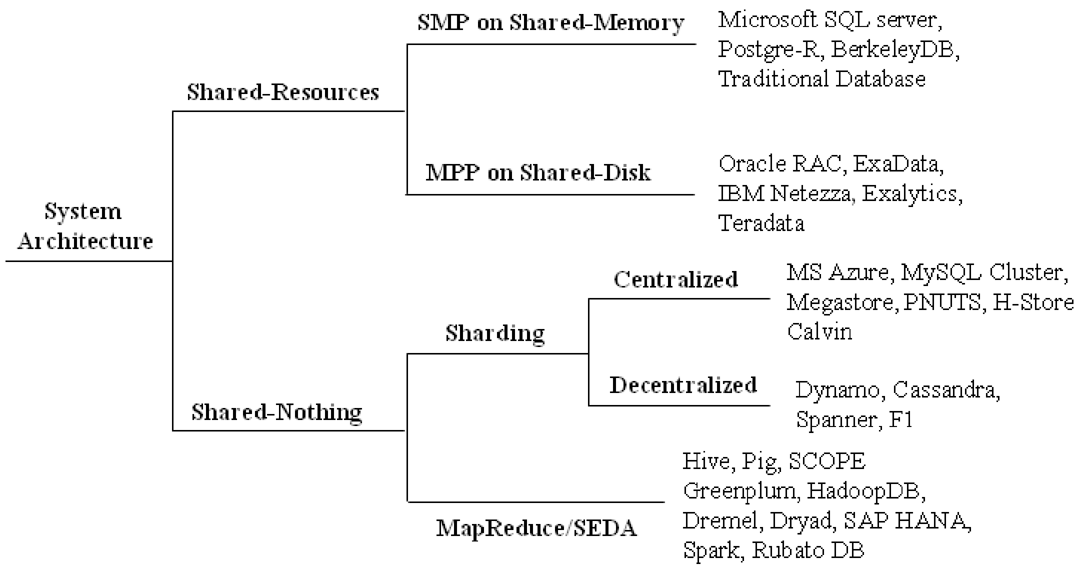

System Architecture
Shared-Resources
SMP on Shared-Memory
Microsoft SQL server, Postgre-R, BerkeleyDB, Traditional Database
MPP on Shared-Disk
Oracle RAC, ExaData, IBM Netezza, Exalytics, Teradata
Shared-Nothing
Sharding
Centralized
MS Azure, MySQL Cluster, Megastore, PNUTS, H-Store Calvin
Decentralized
Dynamo, Cassandra, Spanner, F1
MapReduce/SEDA
Hive, Pig, SCOPE Greenplum, HadoopDB, Dremel, Dryad, SAP HANA, Spark, Rubato DB

Figure 2.6: Taxonomy of system architecture

Microsoft Azure server [Campbell *et al.*, 2010] is built on Microsoft SQL Server and uses centralized topology over the shared-nothing infrastructure. This architectural approach is to inject the specific cluster control with minimal invasion into the Microsoft SQL Server code base, which retains much of the relational features of MS SQL Server. To enhance the scalability, Azure also assembles multiple logical databases to be hosted in a single physical node, which allows multiple local database instances to save on memory for the internal database structures in the server.

MySQL Cluster[17] applies a typical sharding on shared-nothing architecture based on MySQL. Data is stored and replicated on individual data nodes, where each data node executes on a separate server and maintains a copy of the data. MySQL Cluster automatically creates node groups from the number of replicas and data nodes specified by the user. Each cluster also specifies the central management nodes.

H-Store [Kallman *et al.*, 2008] is a highly distributed relational database that runs on a cluster of main memory executor nodes on shared-nothing infrastructure. H-Store provides an administrator node within the cluster that takes a set of compiled stored procedures as inputs.

[17]MySQL Cluster Architecture Overview. MySQL Technical White Paper, 2005.

Megastore [Baker *et al.*, 2011] is a higher layer over Bigtable. Megastore blends the scalability of Bigtable with the traditional relational database. Megastore partitions data into entity groups, providing full ACID semantics within groups, but only limiting consistency across them. Megastore relies on a highly available and persistent distributed lock service for master election and location bootstrapping.

Yahoo! PNUTS [Cooper *et al.*, 2008] is a massively parallel and geographically distributed system. PNUTS uses a publish/subscribe mechanism where all updates are firstly forwarded to a dedicated master, and then the master propagates all writes asynchronously to the other data sites.

Calvin [Thomson *et al.*, 2012] is designed to serve as a scalable transactional layer above any storage system that implements a basic distributed non-transactional storage. Calvin organizes the partitioning of data across the storage systems on each node, and orchestrates all network communication that must occur between nodes in the course of transaction execution with optimized locking protocol.

System above all elect and utilize certain logically central nodes to manage the coordination of the whole cluster, thus they all belong to the centralized topology category. Dynamo [DeCandia *et al.*, 2007], Cassandra [Lakshman and Malik, 2010] and Spanner [Corbett *et al.*, 2012] opt the symmetric structure on the decentralized topology over the centralized one based upon the understanding that symmetry in decentralization can simplify the system provisioning and maintenance. Systems with the decentralized topology basically employ a distributed agreement and group membership protocol to coordinate actions between nodes in the cluster.

Dynamo [DeCandia *et al.*, 2007] uses techniques originating in the distributed system research of the past years such as DHTs [Gummadi *et al.*, 2003], consistent hashing [Karger *et al.*, 1997], quorum [Alvisi *et al.*, 2001], etc. Dynamo is the first production of system to use the synthesis of all these techniques [DeCandia *et al.*, 2007].

Facebook Cassandra [Lakshman and Malik, 2010] is a distributed storage system for managing very large amounts of structured data spread out across many commodity servers. Cassandra brings together the data model from the Bigtable and the distributed system technologies from Dynamo.

Spanner [Corbett *et al.*, 2012] is a scalable, multi-version, globally-distributed database system based on the "True Time" API, which combines an atomic clock and a GPS clock to timestamp data so it can then be synchronized across multiple machines without the need of centralized control. F1 [Shute *et al.*, 2013] is built on top of Spanner, which provides extremely scalable data storage, synchronous replication, and strong consistency and ordering properties.

Hive [Thusoo *et al.*, 2009; Thusoo *et al.*, 2010], Scope [Chaiken *et al.*, 2008] and Pig latin [Olston *et al.*, 2008], built on top of Hadoop, compile SQL-like declarative queries into a directed acyclic graph of MapReduce jobs executed on Hadoop. They systematically leverages technologies from both parallel databases and MapReduce framework throughout the software stack.

Spark [Zaharia *et al.*, 2010] introduces resilient distributed dataset that lets applications keep data in memory across queries, and automatically reconstruct data nodes in failure. Its parallel operations fit into the iterative MapReduce which extends the traditional framework to support iterative data analysis. Spark focuses on applications that reuse a set of data across multiple parallel operations. Shark [Xin *et al.*, 2013] is a low-latency system built on Spark, which can efficiently combine SQL engine and machine learning workloads, while supporting fine-grained fault recovery.

Greenplum [Cohen *et al.*, 2009] is a hybrid system that enables to execute write functions in SQL queries across multiple nodes in MapReduce style. It makes the effort for parallel loading of Hadoop data, retrieving data with MapReduce, and accessing Hadoop data by SQL.

HadoopDB [Abouzeid *et al.*, 2009] is built based on the idea of providing Hadoop access to multiple single-node DBMS servers and pushing data as much as possible into the engine. HadoopDB is to connect multiple single-node database systems by using Hadoop as the task coordinator and the network communication layer. Queries are parallelized across nodes using the MapReduce framework.

Dremel [Melnik *et al.*, 2010] uses a multi-level serving tree to execute queries that resemble the SEDA for data process. Each query gets pushed down to the next level in the serving tree, and is rewritten at each level. The result of the query is assembled by aggregating the replies received from the leaf servers at the lowest level of the tree which scan the tablets in the storage layer in parallel.

Dryad [Isard *et al.*, 2007] is based on a direct acyclic graph (DAG) that combines computational vertices with communication channels to form a data flow graph. The vertices of the graph are on a set of available computers, communicating through files, TCP pipes, and shared-memory FIFOs. Dryad schedules vertices to run simultaneously on multiple computers for parallelism. The arbitrary execution data flow through the communication channel in Dryad is identical to the SEDA.

SAP HANA [Sikka *et al.*, 2012] database is the core of SAP's new data management platform. It introduces the calculation graph model that follows the classical data flow graph principle. The calculation model defines a set of intrinsic operators based on different types of nodes. Source nodes represent persistent table structures or the outcome of other calculation graphs. Inner nodes reflect logical operators consuming one or multiple incoming data flows and produce any number of outgoing data flows.

2.3 Consistency Model

One of the challenges in the design and implementation of big data management systems is how to achieve high scalability without sacrificing consistency. The consistency property ensures the suitable order and dependency of operations throughout the system, helping to simplify application development. However, most large-scale data management systems currently implement a trade-off between scalability and consistency in that strong consistency guarantees, such as ACID [Lewis *et al.*, 2002], are often renounced in favor of weaker ones, such as BASE [Cooper *et al.*, 2008]. In this section, we are going to classify systems according to different consistency levels based on ACID and BASE.

2.3.1 ACID

There are a set of properties that guarantee that database transactions are processed reliably, referred to as ACID (Atomicity, Consistency, Isolation, Durability). Database management systems with ACID properties provide different isolation levels, mainly include serializability, snapshot isolation and read committed [Berenson *et al.*, 1995].

Serializability, the highest isolation level, guarantees that the concurrent execution of a set of transactions results in a system state that would be obtained if transactions were executed serially, i.e., one after the other. It is typically implemented by pessimistic reads and pessimistic writes, achieving the condition that unless the data is already updated to the latest state, the access to it is blocked.

Snapshot isolation is a multi-version concurrency control model based on optimistic reads and writes. All reads in a transaction can see a consistent committed snapshot of the database. A data snapshot is taken when the snapshot transaction starts, and remains consistent for the duration of the transaction. Restrictions such as "first-committer-wins" rule allow snapshot isolation to avoid the common type of lost update anomaly [Bornea *et al.*, 2011].

Read committed, allowing applications trading off consistency for a potential gain in performance, guarantees that reads only see data committed and never sees uncommitted data of concurrent transactions.

If we use the symbol $>$ to represent the stronger relationship among two isolation levels, it is shown that [Berenson *et al.*, 1995]:

$$\textit{serializability} > \textit{snapshot isolation} > \textit{read committed}$$

To provide high availability and read scalability, *synchronous replication* is an important mechanism. With *synchronous replication*, rather than dealing with the inconsistency of the replicas, the data is made unavailable until updates operations are propagated and completed in all or most of replicas. Update operations may be rejected and rolled back if they fail to reach all or a majority of the destination replicas within a given time. When *serializable* consistency is combined with *synchronous replication*, we can achieve *one-copy serializability* [Bornea *et al.*, 2011], in which the execution of a set of transactions are equivalent to executing the transactions in the serial order within only one single up-to-date copy. Similarly, combining *read committed* and *snapshot isolation* with *synchronous replication*, *one-copy read committed* and *one-copy snapshot isolation* can be obtained, respectively [Lin *et al.*, 2005].

2.3.2 BASE

The ACID properties work fine for horizontally scalable, relational database clusters. However, they may not well fit in the new unstructured or non-relational, large-scale distributed systems, in which flexible key/value paradigm is favored and the network partition or node failure can be normal rather than rare. Naturally, many large-scale distributed key-value store systems, such as Amazon Dynamo [DeCandia *et al.*, 2007], Yahoo! PNUTS [Cooper *et al.*, 2008] and Facebook Cassandra [Lakshman and Malik, 2010], choose BASE, a consistency model weaker than ACID. The BASE, standing for *Basically Available, Soft-state, Eventually consistent*, can be summarized as: the system responses basically all the time (Basically Available), is not necessary to be consistent all the time (Soft-state), but has to come to a consistent state eventually (Eventual consistency) [Pritchett, 2008].

Various BASE consistency models have been specified, and thus we first categorize these models and present multiple system implementations to demonstrate different levels of consistency model.

Eventual Consistency

Eventual consistency, one of the fundamental requirements of BASE, informally guarantees that, if no new updates are made to a given data item, eventually all accesses to that data item will return the last updated value.

Even though a system with eventual consistency guarantees to "eventually" converge to a consistency state, it increases the complexity of distributed software applications because the eventual consistency does not make safety guarantees: an eventually consistent system can return any value before it converges [Pritchett, 2008; Lloyd *et al.*, 2011]. Eventual consistency may not provide a single image system since it makes no promise about the time intervals before the convergence is reached. In addition, the value that eventually achieved is not specified. Thus, additional restriction is required from applications to reason the convergence [Roh *et al.*, 2011; Shapiro *et al.*, 2011; Vogels, 2008].

Causal Consistency

Causality is described as an abstract condition that ensures execution in a cluster agrees on the relative ordering of operations which are causally related. Conditions of causality based on reads and writes derive from causal memory [Ahamad *et al.*, 1995]. Causal consistency guarantees the relative ordering of read and write operations that are causally related [Lloyd *et al.*, 2011; Bailis *et al.*, 2013*b*]. Causality is described as an abstract condition that ensures execution in a cluster agrees on the relative ordering of causally related operations.

Implementation of causal consistency usually involves dependency tracking [Bailis *et al.*, 2013*b*; Lloyd *et al.*, 2011; Lloyd *et al.*, 2013]. The dependency order defines the relative ordering of read and write operations that are causally related. It requires that reads respect the order of causally related writes. Under causality, all operations that could have influenced one operation must be visible before the operation takes effect. Dependency tracking associated with each operation is employed to record meta-information for reasoning about the causality. Each process server reads from their local data items and determines when to apply the newer writes to update the local stores based on the dependency tracking.

Ordering Consistency

Instead of merely ensuring partial orderings between causality dependent operations, ordering consistency is an enhanced variation of causal consistency ensuring global ordering of operations. Ordering consistency provides the monotonicity guarantee of both read and write operations to each data item.

(a) The "monotonic writes" ensures write operations being applied in the identical order on all nodes.

(b) The "monotonic reads" ensures that reads only see progressively newer versions of data on each node.

The "monotonic writes" guarantee can be enforced by ensuring that write operation can be accepted only if all writes made by the same user are incorporated in the same node [Saito and Shapiro, 2005]. It can be achieved by designating one node as the primary node for every record; and then all updates to that record are first directing to the primary node. The primary node orders operations by assigning them monotonically increasing sequence numbers. All update operations, together with their associated sequence numbers, are then propagated to non-primary nodes by subscribing them to a queue ensuring updates are delivered successfully. In the case that the primary node fails, one of the non-primary nodes is elected to act as the new primary node [Cooper *et al.*, 2008; Lloyd *et al.*, 2011; Burckhardt *et al.*, 2012].

From the analysis above, causal consistency is stronger than eventual consistency. Further, it is not difficult to see that ordering consistency is stronger than causal consistency in that a system that guarantees ordering consistency also guarantees causal consistency, but not vice versa, since the causal consistency does not prevent conflicting updates [Lloyd *et al.*, 2013]. If we use the symbol $>$ to represent the stronger relationship among two consistency models, the following demonstrates that all the three consistency models form a linear order:

$$\textit{ordering consistency} > \textit{causal consistency} > \textit{eventual consistency}$$

More generally, if we consider each operation in BASE as a single operation transaction[18], the operation schedule in BASE can have an equivalent schedule in the view of single operation transaction in ACID. Considering the bounded staleness [Bailis *et al.*, 2012*a*; Cipar *et al.*, 2012] for values observed by read, the ordering consistency permits possibly stale of data for read at low cost; however, the read committed guarantees that every read observes the most recent and consistent value of data committed before the start of the transaction. We may conclude the stronger relationship among all consistency models in both ACID and BASE as:

$$\textit{serializability} > \textit{snapshot isolation} > \textit{read committed} > \textit{ordering consistency} >$$
$$\textit{causal consistency} > \textit{eventual consistency}$$

2.3.3 Consistency Model Taxonomy

Based on the discussion of consistency models, we categorize the implementation of different systems into the taxonomy as shown in Figure 2.7. The classification is based on our ensuing analysis.

Spanner [Corbett *et al.*, 2012; Shute *et al.*, 2013], Megastore [Baker *et al.*, 2011] and Spinnaker [Rao *et al.*, 2011] provide one-copy serializability with a two-phase commit and Paxos-based protocol [Chandra *et al.*, 2007]. Megastore and Spinnaker provides serializable pessimistic transactions using strict two-phase locking protocol.

[18]By the view of single operation transaction, we mean "start transaction" and "commit" are added before and after each and every database operation.

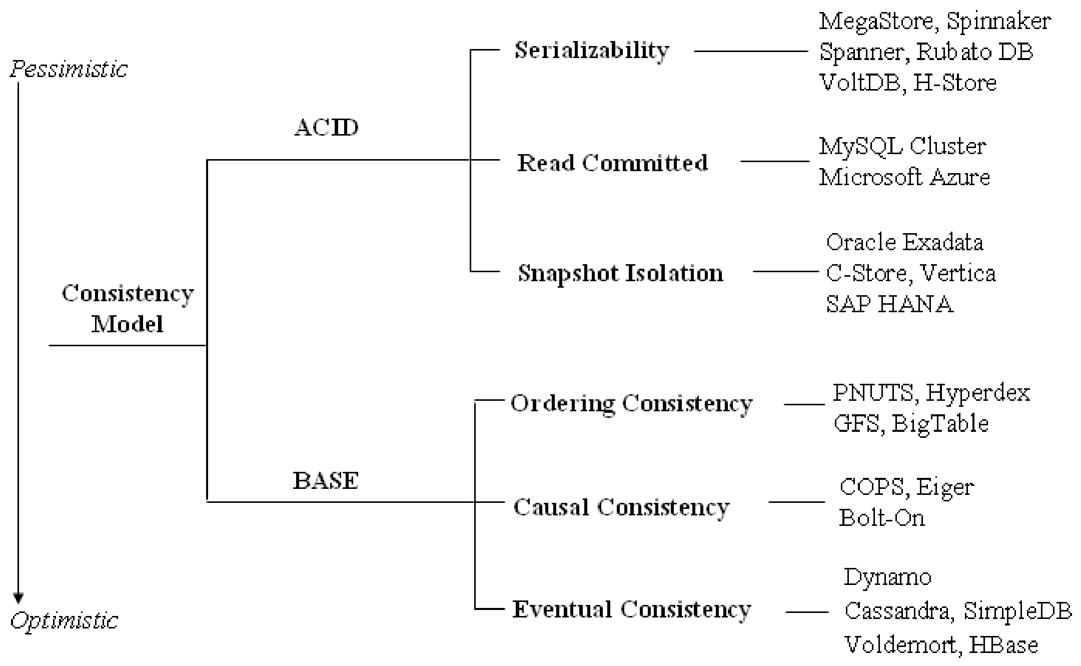


Figure 2.7: Taxonomy of Consistency Model

Spanner adopts strong timestamp semantics. Every transaction is assigned a commit timestamp, and these timestamps allow Spanner to correctly determine whether a state is sufficiently up-to-date to satisfy a read. Paxos protocol [Chandra *et al.*, 2007] ensures that data will be available as long as the majority of the replicas are alive. To support transactions across multiple sites, the two-phase commit protocol and Paxos are usually combined, such as MDCC [Kraska *et al.*, 2013], 2PC-PCI [Lloyd *et al.*, 2013], Paxos-CP [Patterson *et al.*, 2012], and a group of engines have been implemented, such as Chubby [Burrows, 2006] and ZooKeeper [Hunt *et al.*, 2010].

VoltDB[19] and H-Store [Kallman *et al.*, 2008; Jones *et al.*, 2010; Stonebraker *et al.*, 2007] support the SQL transaction execution through the stored procedure. By initiating a global order before execution, all nodes can asynchronously execute the stored procedures serially to completion with the same order. Furthermore, H-Store and VoltDB perform the sequential execution in a single-threaded manner without any support for concurrency. The combination of above mechanisms makes the transaction execution in those systems resemble a single operation call.

MySQL Cluster[20] and Microsoft Azure [Campbell *et al.*, 2010] combined tradi-

[19]https://voltdb.com/

[20]MySQL Cluster Architecture Overview. MySQL Technical White Paper, 2005.

tional read committed with the master-slave mode synchronization. Exadata[21] isolates read-only transactions using snapshot isolation. High water mark with low-overhead mechanism is introduced for keeping track of the value in multi-replica environment in C-Store [Stonebraker *et al.*, 2005]. SAP HANA [Sikka *et al.*, 2012] relies on MVCC as the underlying concurrency control mechanism to synchronize multiple writers, and provide distributed snapshot isolation.

HyperDex [Escriva *et al.*, 2012] provides ordering consistency with a chaining structure, in which nodes are arranged into a value-dependent chain. The head of the chain handles all write operations and dictates the total order on all updates to the object. Each update flows through the chain from the head to the tail, and remains pending until an acknowledgement of commit received from the tail. Read operations can be dealt with by different nodes as long as the latest committed value of data can be obtained on that node.

Yahoo! PNUTS [Cooper *et al.*, 2008] provides a per-record timeline consistent model that preserves ordering consistency. PNUTS introduces a pub/sub message broker, which takes charge of receiving updates from a master node and sending them to other nodes in an identical sequence order.

GFS [Ghemawat *et al.*, 2003] and Bigtable [Chang *et al.*, 2008] both use Chubby [Burrows, 2006], a distributed locking mechanism for distributed node coordination. They rely on Chubby with a lease agreement to apply mutations to each chunk in the same order. The global mutation order is defined by the lease grant order.

COPS [Lloyd *et al.*, 2011; Burckhardt *et al.*, 2012] and Eiger [Lloyd *et al.*, 2013] track dependencies on versions of keys or operations to enforce causal consistency. An operation does not take effect until verifying that the operation's dependencies are satisfied.

Bolt-on [Bailis *et al.*, 2013*b*] provides a shim layer that upgrades the eventual consistency of an underlying general-purpose data store to the causal consistency for the application. Bolt-on sets a generalized, declarative specification of causal cut as the criteria to determine which writes are made visible to clients.

Dynamo [DeCandia *et al.*, 2007], Voldemort[22] and Cassandra [Lakshman and Ma-

[21]Technique overview of the Oracle Exadata Database Machine and Exadata Storage Server. Oracle White Paper, 2012.

[22]http://project-voldemort.com/

lik, 2010] provide eventual consistency for allowing applications with "always writeable" property, that is, write operations can always be accepted by any node. Vector clock, also named as version vector, is associated with data to determine the eventual consistent state during reconciliation.

2.4 Implementation Principles for Large-Scale Data Management System

So far, we have built the taxonomies for the state of the art large-scale data management systems based on the data model, the system architecture and the consistency model. In this section, we will discuss and analyze the scalability limitation for different designs and implementations, and provide our principles for the implementation of the next generation large-scale data management systems.

2.4.1 Data Model Integration

To achieve high scalability, systems need to distribute data to different nodes. The simplest way for the data distribution is to deploy individual tables at different nodes. However, since concurrent transactions usually access different portions of one table, thus the table partition can improve the performance by parallel execution on a number of nodes. In addition, when processing a single query over a large table, the response time can be reduced by distributing the execution across multiple nodes [Cohen *et al.*, 2009]. Partitioning can be either horizontal or vertical, and systems require a close integration between data model and data partition.

Physical level. For the row-oriented layout, horizontal partitioning is properly used where each partition contains a subset of the rows and each row is in exactly one partition. For column-oriented or hybrid layout, vertical or mixed partitioning can be applied. The fine-grained mixed partitioning is often implemented by vertically partitioning columns of a table into frequent and static column groups. Columns frequently accessed together are clustered into the same frequent column group, while columns seldom accessed are gathered into static column groups [Grund *et al.*, 2010; Jones *et al.*, 2010]. In addition, columns with large-size data are separated independently to take the advantage of the compression benefits of column stores [Abadi *et al.*, 2006].

Conceptual level. Regarding to the conceptual schema, numerous NoSQL high scalable systems such as key-value stores [DeCandia *et al.*, 2007] and Bigtable-like stores [Chang *et al.*, 2008] represent a recent evolution of making trade-off between scalability and schema complexity. These NoSQL systems adopt variant of schema-less unstructured data for large-scale data applications. Atomicity and isolation are supported at the granularity of a single key-value pair, that is, an atomic operation is supported only for individual key-value pairs. Restricting data accesses to a single-key limits the flexibility of operating at a much finer granularity. Since a single key-value pair is never split across compute nodes, data manipulation is restricted within a single compute node boundary and thus obviates the need for multi-node coordination and synchronization. As a result, systems can only horizontally partition key-value pairs to scale out. The rationale is that even though there can be potentially millions of requests, the requests are generally distributed throughout the data set.

However, some NewSQL systems relatively to NoSQL seek to provide the high scalability and throughput same as NoSQL while still preserving the high level of structured data model [Kallman *et al.*, 2008; Baker *et al.*, 2011; Corbett *et al.*, 2012; Cohen *et al.*, 2009]. To achieve high scalability for a structured data store, the partitioning must be based on the relational schema and query workloads in order to minimize the contention. Inappropriate partitioning may cause data skew. The skewed data will decline the response time and generate hot nodes, which easily become a bottleneck throttling the overall performance and scalability [Kossmann *et al.*, 2010; Xu *et al.*, 2008]. For example, to obtain the optimal partitioning, the schema of OLTP workload always transits to a tree structure. Tuples in every descendent table are partitioned according to the ancestor that they descend from. As a consequence, data accessed within a transaction will be located in the same data node [Grund *et al.*, 2010; Jones *et al.*, 2010].

Based on the analysis above and the taxonomy, we believe that:

1. The data model with lower conceptual level can simplify the design and implementation for scale-out capability, but data model with higher conceptual level is not an obstacle for scalability as long as an optimal partitioning strategy is

applied according to the schema.

2. The hybrid storage is the most flexible to leverage the advantages of row stores and column stores. And hybrid storage facilitates the close integration for data modeling and data partitioning.

2.4.2 Architecture Scalability

Systems based on shared-resources infrastructure can only be scaled up with inevitable bound due to resources contention and limitation. For example, the internal bus bandwidth, the number of CPU cores, the hardware coordination mechanism [Poess and Nambiar, 2005], the context switches [Stonebraker *et al.*, 2007; Johnson *et al.*, 2009], all can inhibit scalability.

In the sharding on shared-nothing architecture, local resources serve local processors; thus it overcomes the disadvantage of shared-resource infrastructure. The shared-nothing design is intended to support scaling out smoothly by involving new computation nodes. The centralized topology uses one dedicated centralized coordinator for managing system-wide membership state. Thus, it may suffer the single point failure and the central node is susceptive to become the bottleneck. The scalability of the decentralized peer-to-peer topology can excel that of the centralized one [DeCandia *et al.*, 2007; Lakshman and Malik, 2010].

Though systems on sharding on shared-nothing architecture are desired to achieve ultimate scalability, and have been proven to scale really well into tens of hundreds of nodes. However, there are very few known data management system deployments with thousands of nodes [Abouzeid *et al.*, 2009; Shute *et al.*, 2013; Melnik *et al.*, 2010; Isard *et al.*, 2007].

SEDA/MapReduce architecture is able to scale to thousands of nodes due to its superior scalability and flexibility. In particular,

(a) The architecture facilitates the use of shared-nothing infrastructure as a scalable platform for applications, which is easy to scale out the critical components by adding more computing resources. Computing resources can be distributed flexibly to wherever more computation power is needed to eliminate bottleneck.

(b) The decomposition of a complex database management system into various stages connected by queues not only enables modularity and reuse of stage modules, but also allows two types of parallel data processing: pipeline parallelism and partition parallelism.

(c) Since a data flow is distributed not only between multiple server instances (usually running on different physical nodes) but also between different staged modules, multiple identical stages can be executed on multiple cluster machines with data replication in order to achieve high availability for fault tolerance.

(d) The architecture easily supports data partition. Different partitions can be distributed to different staged modules on multiple computing nodes. Different stage modules are communicated by passing operation. The downstream module can start consuming operation when the producer module passes the operation.

A large number of system applications based on this architecture are such good examples:

1. Google uses the MapReduce framework internally to process more than 20PB datasets per-day, achieving the ability to sort 1 PB data using 4,000 commodity servers [Dean and Ghemawat, 2008].

2. Hadoop at Yahoo! is assembled on 3,000 nodes with 16PB raw disk [Shvachko *et al.*, 2010].

3. At Facebook, Hive [Thusoo *et al.*, 2009] forms the storage and analytics system infrastructure that stores 15PB data and process 60TB new data everyday with thousands of nodes [Thusoo *et al.*, 2010].

4. Microsoft Dryad conducts data mining style operations to tens of perabytes data using a cluster of around 1800 computers [Isard *et al.*, 2007].

5. Google Dremel manages trillion-record, multi-perabyte datasets, running on 1000-4000 nodes with near-linear scalability [Melnik *et al.*, 2010].

Based on the above empirical evidence and the system taxonomy, we can have that the scalability capacity of these different architectures can be ordered as:

SEDA/MapReduce >> Sharding Decentralized > Sharding Centralized >>
Shared-Disk MPP > Shared-Memory SMP.

Thus, the SEDA/MapReduce achitecture is the most suitable for large-scale data management systems.

2.4.3 Scalable Consistency Implementation

We propose a consistency model taxonomy in which the model in a higher level provides stricter guarantees than the model in a lower level. Now we discuss the relationship between consistency and scalability, and especially how the implementation of different consistency models will affect the scalability.

The common implementation to provide serializability is based on distributed two-phase locking (2PL) protocol [Baker *et al.*, 2011; Rao *et al.*, 2011]. However, the locking-based protocol adds overhead to each data access due to the manipulation to acquire and release locks, and it limits concurrency and scalability in case of conflicting accesses, and adds overheads due to deadlock detection and resolution [Larson *et al.*, 2011; Özsu and Valduriez, 2011; Weikum and Vossen, 2001]. Another similar pessimistic concurrency control protocol implementation is based on distributed serialization graph testing (SGT), which characterizes conflict serializability via the absence of cycles in the conflict graph [Özsu and Valduriez, 2011]. The limitation of this implementation is closely related to the problem of testing a distributed graph for cycles, which also arises the deadlock detection issue. Thus, transactions executed by a distributed large-scale database should not cause distributed deadlocks that are rather difficult and expensive to deal with.

Optimistic protocol is lock-free assuming that conflicts between transactions are rare. However, to guarantee that the validation phase can produce consistent results, a global order checking is required which will degrade performance heavily in large-scale distributed data systems [Corbett *et al.*, 2012]. Additionally, in the presence of slower network connection, more transactions may crowed into the system causing excessively high chances of rollbacks [Larson *et al.*, 2011]. Therefore, the timestamp-based concurrency control is the most suitable for large-scale data management systems, and Google Spanner [Corbett *et al.*, 2012] based on True Time is such a good example.

The communication latency caused by various protocol implementations (e.g. two-phase commit, three-phase commit, Paxos, etc.) can limit the scalability due to the overhead caused by multiple network round-trips [Helland, 2007]. Though a protocol implementation performs well within a small scale cluster, however, it may severely limit the scalability of large-scale systems, since the availability and coordination overheads become worse as the number of nodes increases [Yu and Vahdat, 2002; Yu and Vahdat, 2006].

Constraining the scope of transactions is one typical way to minimize the high-latency communication overhead. The restriction of transactions alleviates the transaction coordination protocol and reduces message delays. A list of systems such as Azure [Campbell *et al.*, 2010], MySQL Cluster[23], Megastore [Baker *et al.*, 2011], Sinfonia [Aguilera *et al.*, 2009] and H-Store [Kallman *et al.*, 2008; Stonebraker *et al.*, 2007] only support restricted transactions that can be executed in parallel to completion without requiring communication with other repositories or any commit vote phase. This restrictive scope is reasonable for the applications where data can be well deployed, so that distributed transactions will be very rare in such cases.

The weak consistency model like BASE can achieve high scalability much easier than the strong consistency model like ACID. But the strong consistency model does not hinder the scale-out capability as long as a proper implementation (or reasonable restriction) is developed.

2.5 Chapter Summary

In this chapter, we have investigated, categorized and studied several critical aspects of large-scale data management systems. These systems have several unique characteristics mainly including scalability, elasticity, manageability, and low cost-efficiency. We have first enumerated various data models on physical layouts and conceptual representations. And then, we focused on the design and implementation of system architectures. We have developed architecture taxonomies for prevailing large-scale database systems to classify the common architecture designs, and provide compar-

[23]MySQL Cluster Architecture Overview. MySQL Technical White Paper, 2005.

ison of the capability for scale-out. We then compared two categories of the consistency models and classified prevailing systems according to the respective taxonomies. With these taxonomies, we have gained insights into the trade-off between consistency and scalability. Based on our taxonomies and characterization, we also identify the principles for implementation of large scale data management systems including:

- The hybrid storage layout can facilitate the close integration of data modeling and data partitioning.
- The SEDA/MapReduce architecture is the optimal to achieve high scalability.
- The timestamp-based protocol is the most suitable for the implementation of consistency models to scale out.
- The strong consistency model is not an obstacle for scalability with a proper implementation (or reasonable restriction).

Chapter 3

Overview of Rubato DB

"Music is the art of thinking with sounds."

- Jules Combarieu.

3.1 Features in Rubato DB

Based on the principles outlined in the last chapter, we implement a scalable NewSQL database management system-Rubato DB [Yuan *et al.*, 2014; Yuan *et al.*, 2015], supporting relational data model for data-centric applications. Rubato DB is implemented with the following distinguished features:

- **F1**: Rubato DB applies a novel grid architecture, based on the staged event-driven architecture (SEDA), proposed by Welsh for parallel web servers [Welsh *et al.*, 2001].
- **F2**: Rubato DB uses a hybrid data partitioning based on application semantics and query workloads.
- **F3**: Rubato DB introduces a new formula based protocol for distributed concurrency control.
- **F4**: Rubato DB supports various consistency guarantees from ACID to BASE.

One of the key challenges in designing and developing NewSQL database management system using a collection (cluster) of commodity servers is how to distribute large volume of workloads and data into tens or hundreds of commodity servers with smaller dedicated DAS devices attached. It is **F1** and **F2** that enable Rubato DB to resolve this problem.

Another key challenge is developing a suitable concurrency control protocol for thousands of concurrent users that are accessing data distributed over commodity servers. We have developed a distributed formula protocol for concurrency control under the staged architecture (**F3**, **F4**), which guarantees the consistency of transactions processed by Rubato DB.

The *formula protocol for concurrency* is a variation of the multi-version timestamp Concurrency Control Protocol [Thomas, 1984], which reduces the overhead of conventional implementation with the following mechanisms:

- Instead of using multiple versions of updated data items, the protocol uses simple formulas stored in memory (associated with the updated data items).
- Caching and delaying the transaction operations with formulas before committing.
- A dynamic time-stamp ordering is used to increase the concurrency and reduce unnecessary blocking.

In general, the design and implementation of Rubato DB combines and extends the ideas as following:

> hybrid data partitioning, staged-event driven architecture (SEDA) [Welsh *et al.*, 2001], data-intensive MapReduce framework [Dean and Ghemawat, 2008], and distributed timestamp-based concurrency control;

and using innovative technologies such as

> formula based protocol, software instruction, lazy loading, dynamic timestamp ordering, multiple communication protocols.

3.2 SQL Engine Design

The main architectural components of Rubato DB are depicted in Figure 3.1. We firstly give a brief overview of the several essential Rubato DB components as below.

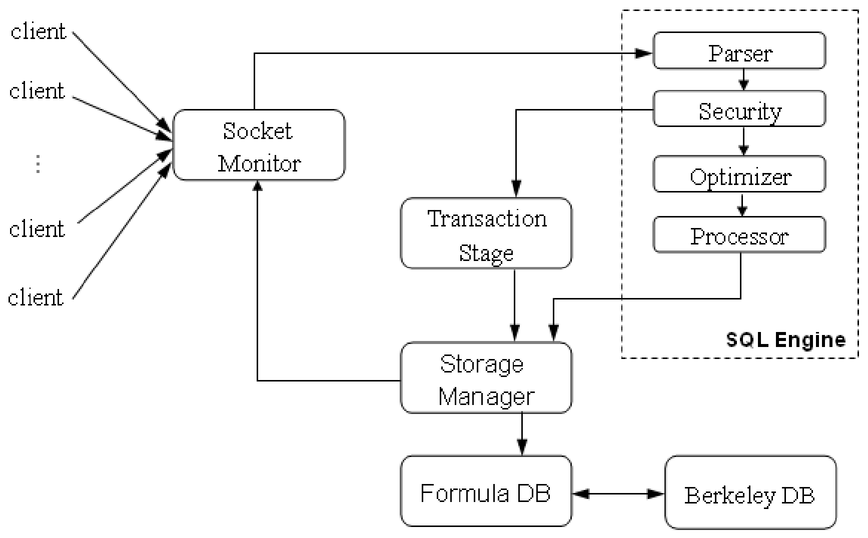


Figure 3.1: RubatoDB system architecture

Socket Monitor Rubato DB runs on a collection of servers as one database management system, with a single socket monitor to establish and record the connection states for all client requests. The socket monitor which the clients link to, hides the logic of connection management and routing, and abstracts the system's dynamics from the client applications while maintaining uninterrupted connections to the system. Multiple socket monitors can be deployed in Rubato DB to distribute the workloads of massive client's requests, in order to reduce the risk of becoming bottleneck and improve the availability for socket monitor. Also note that special mechanism is required to generate global coherent timestamps across multiple socket monitors, such as time master tracking mechanism [Hochschild *et al.*, 2015].

The socket monitor adopts a loading control schema, called *lazy loading*, to reduce data contention and to minimize unnecessary rollbacks. The socket monitor maintains two lists of clients: a list of active clients with active transactions, and a list of requesting clients whose requests are waiting to be evaluated. The socket monitor will guarantee that the waiting transaction is among the oldest 20% ones base on the following practical principle: if all the current requests have higher potential to conflict with other requests, the system should rather wait awhile for new requests with lower conflict potential to arrive.

SQL Engine Rubato DB's SQL engine is used to process all the SQL queries, including aggregate functions and nested queries, updates, and all other requests according to SQL2003. Rubato DB's SQL engine is implemented using a staged architecture, which has been introduced and studied for various applications, such as Dynamic Internet Servers [Welsh *et al.*, 2001] and high-performance DBMSs [Harizopoulos and Ailamaki, 2003]. The basic idea of this architecture is that a system is constructed as a network of staged modules connected with explicit queues. A staged module (or stage) is a self-contained module consisting of an incoming event (request) queue, a thread pool and an event handler [Welsh *et al.*, 2001].

The SQL engine is composed as a set of such staged modules with an incoming request queue. Threads within each staged module operate by pulling a sequence of requests, one at a time, off the input queue, invoking the application-supplied event handler (e.g. parser, optimizer, query, update, etc) to process requests, and dispatching outgoing results by enqueuing them on the request queues of outgoing staged module, located either in the same server or another server within LAN. Each request to the system will be processed in a relay-style by being passed from one staged grid module to the next one until it is completed. Both parallelism and pipeline execution are supported by the engine. A set of software instructions are employed to carry the operation's backpack with its private state and data. The instruction with a uniform format is the only packet flowing through different staged modules. The staged architecture provides a satisfactory design for a scalable grid (or distributed) database management system, since the staged modules can be easily arranged to run on various grid servers.

3.3 Transaction Manager

The transaction manager, consisting of *Transaction stage* and *Formula DB*, is responsible for coordinating data access on multiple nodes based on a novel formula protocol for concurrency (FPC) to ensure serializability. It is the transaction manager that performs all the transactional operations, including pre-commit (necessary for distributed concurrency), commit, and rollback.

The FPC is a novel implementation of the classical Multi-version Timestamp

Concurrency Control Protocol [Thomas, 1984], with two distinct features:

1. Instead of using multiple versions of updated data items, FPC uses formulas stored in memory (associated with the updated data items) to represent the multiple values of updated data items, which will reduce overhead of storing multiple versions of data replicas.

2. The timestamp ordering of transactions may be altered to allow a transaction with older timestamp to read the data item updated by a later transaction, as long as the serializability is respected, which will increase the degree of concurrency.

The two parts of the transaction manager perform their respective responsibilities:

1. *Transaction stage.* A dedicated stage grid module that is located in every grid server and is responsible for all the basic transactional operations including *pre_commit*, *commit* and *rollback* of the transaction manager.

2. *Formula DB.* A thread-free layer on the top of the Berkeley DB such that all disk accesses in Rubato DB are through Formula DB.

Three different consistency guarantees are supported by Rubato DB including:

1. ACID. The strongest end of the consistency spectrum for the transactional functionalities.

2. BASE. The most notable weak consistency model used by NoSQL systems [Cooper *et al.*, 2008; DeCandia *et al.*, 2007; Lakshman and Malik, 2010]. The BASE can be summarized as: the system responses basically all the time (Basically Available), is not necessary to be always consistent (Soft-state), but has to come to a consistent state eventually (Eventual consistency) [Pritchett, 2008].

3. BASIC. Rubato DB is not limited to merely ACID that is too strong and BASE that is too weak, but rather supports BASIC, a spectrum between these two extreme. BASIC stands for Basic Availability, Scalability, and Instant Consistency [Wu *et al.*, 2014]. BASE and BASIC provide different choices between

the model that is faster but requiring extra efforts to deal with inconsistent results and the model that delivers consistent results but is relatively slower with higher latency.

3.4 Storage Manager

Numerous systems such as key-value stores [Ghemawat *et al.*, 2003; DeCandia *et al.*, 2007] and Bigtable-like stores [Chang *et al.*, 2008; Lakshman and Malik, 2010] represent a recent evolution in building infrastructure by making trade-off between scalability and functionality. They adopt variant of the unstructured or semi-structured data. Since Rubato DB as a NewSQL database system aims for data-centric applications, the data model is still based on the widely-accepted relational data model where data is organized into tables of records with attributes according to pre-defined and flexible schema.

Obtaining performance requires a close integration between data model and physical layout. Rubato DB's storage manager adapts a hybrid layout to support combination of row-oriented and column-oriented storage model in order to take advantage of the data continuity of values in each column, thus the I/O wait for queries frequently accessing a subset of attributes in the table schema can be reduced significantly.

To achieve linear scalability, extra effort on an appropriate system configuration is also required. The configuration is highly dependent on the actual data and queries in the workload. An improper configuration can cause the performance decline greatly. A poor data partitioning could cause the problem of redistribution skew [Xu *et al.*, 2008] or fluctuating workload [Kossmann *et al.*, 2010]. The skewed data not only declines response time, but also generates hot nodes, which easily become a bottleneck throttling the overall performance.

To obtain maximum performance, when a table is created according to the schema, the storage manager provides facilities for users to specify fine-grained hybrid storage partitioning based on application semantics and query workload. The fine-grained hybrid schema integrates row-oriented and column-oriented layouts in the granularity of individual tables.

The varieties of data partitioning depend on how the tables are accessed. For columns accessed as a part of sequential scans, narrow vertical partitions can perform better due to cache locality; whereas, horizontal partitions perform better for frequently accessing many of fields of a row. With the analysis on different schemes of data partitioning, we choose to split the tables with hybrid partitioning, and deploy subtables into different nodes with beneficial storage model. The data partitioning task is done in two aspects, as depicted in Figure 3.2.

The storage manager supports grid partitioning based on the tree-based schema. A schema has one root table whose primary key acts as the partitioning key. A root table can however have multiple descendant tables. Every descendant table in the schema will have the root table's key as a foreign key. The tree structure implies that corresponding to every row in the root table, there are a group of related rows in the descendant tables. Such grid partitioning on tree-based schema can work well for most OLTP applications based on the fact that transactions only access data in the same group descended from the root table key. If numerous transactions violate such boundary, the tree-based schema partitioning will not be able to obtain the best performance. There are also some more complex mechanisms that deal with partitioning merely based on real workloads, such as set merging [Agrawal *et al.*, 2004] and graph-based partitioning [Curino *et al.*, 2010]. However, compared with tree-based schema partitioning, these mechanisms may induce overheads due to the complexity. In addition, the grid partitioning based on the tree schema structure allows efficient splitting and merging of partitions. Even though such grid partitioning on tree-based schema does not encompass the entire spectrum of OLTP applications, most of them either have such an inherent tree-based schema pattern or can be easily adapted to it [Bernstein *et al.*, 2011].

In addition, some tables are look-up tables that are mostly read-only. Since such tables are not updated frequently, they are replicated on all the nodes.

The column partition in the storage manager enables users to store a single relation as a collection of disjoint (none-key columns) vertical partitions of different groups. Columns frequently accessed together are clustered into the same frequent column group, while columns seldom accessed are categorized into static column groups. Compression can be applied to take the benefits of column-oriented layout.

Storage manager uses Formula DB for all disk accesses to direct attached storages

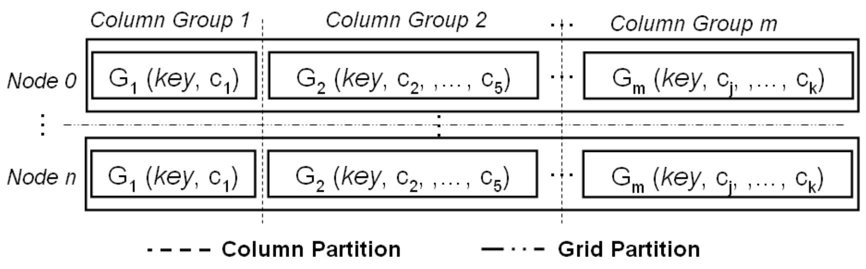


Figure 3.2: Hybrid partition of Storage Manager

(DAS). All tables and their partitions if any are stored on local disk as a Formula DB file.

3.5 Chapter Summary

In this chapter, we describe the overview of Rubato DB, a scalable NewSQL system, showcasing the main feature of scalability with various consistency properties from ACID to BASE, which provides a positive answer to the question on the trade-off between scalability and consistency. In the following chapters, we will introduce detailed information about each features included in Rubato DB.

Chapter 4

A Staged Grid Database

"Every truth passes through three stages before it is recognized. In the first it is ridiculed, in the second it is opposed, in the third it is regarded as self evident."

- Arthur Schopenhauer.

The principle foundation in staged grid database architecture is modularization, encapsulation and communication. By disintegrating and reassembling system components into encapsulated staged modules, different modules can be assigned to self-contained stages. Effective behavior rules for communication are then defined to orchestrate independent staged modules deployed on networked computing nodes into one integrated system.

4.1 A Staged Grid Architecture

The traditional widely adopted, work-centric, thread-parallel and monolithic architecture has the deficiency that could limit the performance when executing excessive data access workloads with changing requirements [Stonebraker *et al.*, 2007]. To harness the available computing power and memory hierarchy for the database architecture, one of the tentative amelioration is introducing the staged-event driven architecture to improve the DBMS effectiveness by optimizing the sheer volume of data access and the unpredictable sequence of memory requests.

The architecture of the scalable grid database we propose is based on the principle behind the staged-event driven architecture that is easy to be integrated into the grid shared-nothing infrastructure.

Similar to the staged architecture, the key insight in redesigning the database architecture is modularization and loose coupling. The basic idea of this architecture is that a system is constructed as a network of staged modules connected with explicit queues. The fundamental unit of processing within a staged architecture is the *staged module* or the *stage* for short. A stage is a self-contained module consisting of an incoming request (also called event) queue, a thread pool, and an event handler. Threads within a stage operate by pulling a sequence of requests, one at a time, off the input queue, invoking the application supplied event handler to process requests, and dispatching outgoing events by enqueuing them on the request queues of outgoing stages.

Now we specify the staged grid architecture by integrating the staged approach into the grid shared-nothing environment. The definition facilitates easy and efficient implementation.

Definition 1 *A **process module** is a software component that operates within that system independent of the operations of the other components.*

Definition 2 *A **staged module** is a software module that interacts with other software components only by means of message queues.*

Definition 3 *A **staged grid module** is a staged module that runs on a grid node. Let **M** be a staged grid module, **data(M,e)** be the state of the staged grid module **M**. The semantic behavior of **M** is specified by two transition rules:*

- *For an input request e, the data transition rule data(M,e): data $\xrightarrow{e}$ data$'$ transforms the given data to data$'$;*
- *For an input request e, the output transition rule output(M,e): data $\xrightarrow{e}$ output creates the output offer based on the given request e and the current state of data.*

Each grid module consists of an input queue storing all the requests to be evaluated, and the module will continuously pull requests off the queue and then executes it. At any given moment different modules are processing multiple requests concurrently. It can be assumed that the request queue in each grid module is reliable and unbounded, which guarantees that the delivered requests are kept in order and can never be lost. A round of computation in a grid module starts when the module pulls the top input request off the queue, and ends when the module enqueues the outgoing event, also called an output offer, into the tail of the input queue of the next stage for further processing.

Definition 4 *A grid module is said to be* ***grid-encapsulated*** *if it accesses only the data stored in the DAS (Direct-Attached Storage) of the node, and does not interact (call) any other modules directly.*

The grid-encapsulation is to specify the data locality: A grid module is grid-encapsulated only if it does not access data stored in the storage devices other than the local disk attached to the grid node. Obviously, a grid-encapsulated stage is easy to develop and efficient to operate, especially for concurrency operations. Since each staged module is encapsulated, meaning that it performs its computation without any specific dependency on other modules. Each single client request visits several different grid nodes during the different phases of the execution, which is transparent to the clients.

Definition 5 *A* ***staged grid architecture*** *is a software system architecture such that:*

(a) The system is constructed as a network of grid encapsulated modules on the shared-nothing infrastructure, i.e., all grid nodes are connected by a (high speed or otherwise) network.

(b) Communication between two modules is by means of the event queues through the network.

(c) When a staged module pushes a request to the next module, the request will always be accepted and kept in the event queue of the destination module.

(d) Each staged module attempts to read requests from its input event queue. The module will wait if no available request can be obtained.

(e) *Each request to the system will be processed in a relay-style by passing the request from one staged grid module to the next one until it is completed.*

The staged grid architecture, as illustrated in Figure 4.1, facilitates the use of shared-nothing infrastructure, which is easy to scale out the critical components by adding more computing resources. The decomposition of a complex system into stages connected by queues not only enables modularity and reuse for each stage, but also allows parallelism and pipelining execution. Different grid stage components are communicated by passing operation. The downstream module can start consuming operation when the producer module passes the operation. Each individual staged grid module is programmed in a usual manner and the operation is coded in the standard format, so that modules at both ends of the queue do not need to have dependencies as long as they can understand the common format.

Multiple identical stages can be executed on multiple cluster machines with data replication in order to achieve high availability. The architecture is also possible to support data partition. Different partitions can be distributed to different staged modules on multiple computing nodes. Particularly, data independent stages, such as parser stage, security stage, can be allocated on arbitary nodes; while the allocation of data dependent stages, such as the query/update stage, are corresponding to the positions of the data. Each update request issued to one data item will be pushed to every update stage that is collocated according to each data replica.

In addition, computing resources can be distributed effectively where more computation power can converge to the bottleneck. The number of operations waiting in the queue can reflects the workload status of each staged grid module. A growing number indicates that module cannot process the work fast enough, thus require additional computing nodes to assist consume operations faster. Observing the length of waiting list and adjusting the number of nodes accordingly, the system can effectively scale out smoothly.

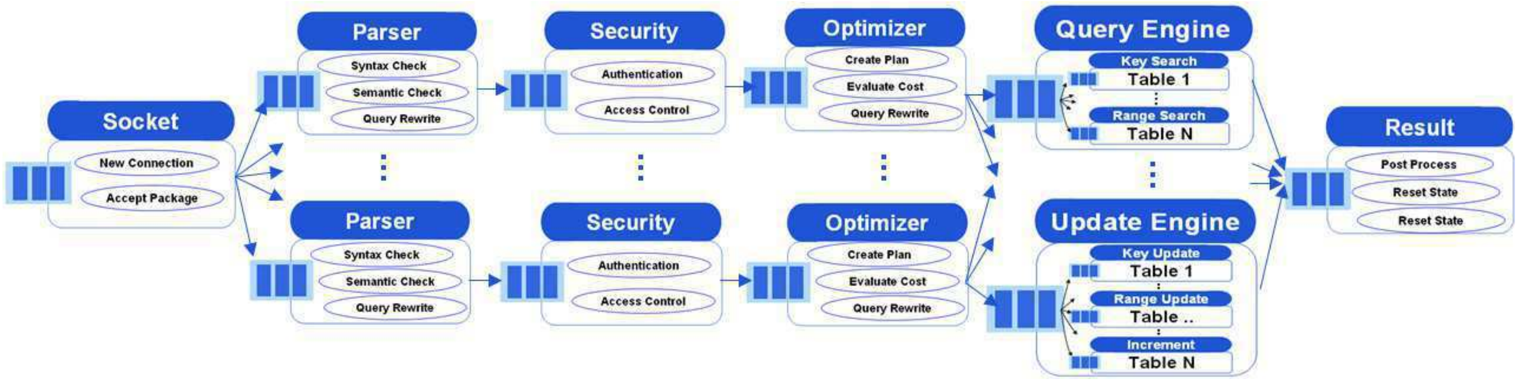


Figure 4.1: The architecture is composed as a set of encapsulated staged grid modules, each of which has its own queue. SQL statements are embedded into an operation with private state and data, and pass through the modules. Edges represent the flow of operations between modules. Each staged grid module is independently managed, and can work in pipeline and in parallel. The execution engine can include a combination of multiple modules to preserve scalability. Multiple identical stages can be executed on data replication in order to achieve high availability. For simplicity, some operation paths and modules have been elided from this figure.

4.2 Implementation of Staged Grid Database

System Deployment

Rubato DB runs on a collection of servers as one database instance with a single socket monitor stage module to establish and record the connection states for all client requests (as illustrated in Figure 3.1 in Section 3.2). Unlike systems such as VoltDB[1], Rubato DB provides a service as a general OLTP store in that all the transactions are received as a sequence of SQL requests, not just a stored procedure. When a request is received by the socket monitor stage, it is assigned a transaction id (i.e., timestamp).

Rubato DB's SQL engine is used to process all SQL queries. The SQL engine is built on top of the staged grid architecture where each component is implemented as a staged module. The essential components of Rubato DB are depicted in Figure 4.2.

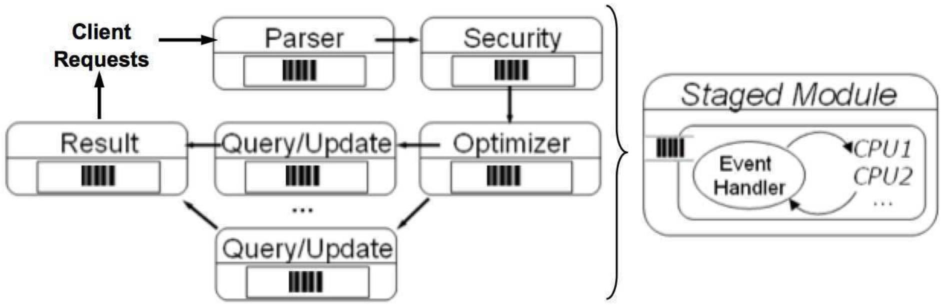


Figure 4.2: Staged Modules of Rubato DB's SQL Engine

Data independent staged modules such as parser and security can be deployed on arbitrary grid nodes, and each node can even consist of multiple such stages for scaling up. Optimizer and processor stage modules (i.e. query stage, update stage) are deployed corresponding to tables per node. Particularly,

1. If a query involves one table on one node, one query stage is set up for each table.

2. If a query involves a join on multiple tables on one node, the join will be carried out by one join stage.

3. If a query involves one table that distributed over different nodes without join,

[1]http://voltdb.com/products/technology

one query stage per node will issue the requests, and one aggregation stage in one of the nodes will collect all result sets and combine them as one result set.

4. If a query involves a join on two or more tables distributed over different nodes, several stages are needed to perform the semi-join based query optimization (not implemented yet[2]).

5. One update stage for each table to perform all update/insertion/deletion operations on each node.

With such deployment, different types of stages can work together for a single query in a relay-style to achieve pipelined execution. The same type of stages associated with different table partitions can also work simultaneously to achieve parallelism. A Transaction stage is deployed on each node, being responsible for coordinating data access on multiple nodes based on a novel formula protocol for concurrency (FPC in Chapter 5).

Software Instruction Set

To facilitate communication among staged modules on different nodes, a set of *software instructions* is introduced to specify all basic operations and/or data packets. Each instruction carries all necessary information required for a request or a data packet, including its *transaction id*, *operation id*, *stage id*, *partition id*, *table id*, *destination node*. Any stage will receive a sequence of incoming instructions from its previous stages, and forwards the processing result, being a result set or a subsequent operation, as an instruction (or a sequence of instructions) to its successive stages for further processing. The instructions with a uniform format are the only packets flowing through stages in the system.

By properly multiplexing the concurrent requests of multiple instructions, there is a potential of increasing instruction reusability for main memory utilization. Rubato DB utilizes a stack as a pool of instructions in each grid node: one instruction is popped up to serve a request when needed, and will be pushed back to the pool after being used.

[2]So far, we can avoid such case by distributing tables with joining on the same node.

Multiple Communication Channels among Stages

Rubato DB utilizes different communication channels among all stages, depending on locations of stages and/or the system resources. Assume the stage S_i is going to send an instruction I into the stage S_j. Three different channels can be used for different cases:

- **I1.** S_i and S_j are in the same node. Since each node maintains one instruction stack pool, it only needs to push the address of I into the operation queue of S_j.

- **I2.** S_i is in Node N_i and S_j is in Node N_j, and N_i, N_j are on a shared-nothing infrastructure. The standard TCP/IP protocol pipes are used to send the content I from N_i into a new instruction I_j in N_j and then push the address of I_j into the operation queue of S_j.

4.3 Determinism Verification

We verify the determinism of the proposed staged grid architecture based on the formalisms of Kahn's PN model [Kahn, 1974] with enhancements that are designed to make it efficiently implementable and preserve the properties including scalability, composability and boundedness.

- Determinism in the system context means that the modeled system will produce the same result every time it is executed. Insufficient restrictions on accesses to shared resources lead to race conditions and can occur with unintended non-determinism. Note that, in the KPN model, we assume that there is neither network latency nor execution node failure. Also, for multiple requests of concurrent clients, we assume that the system will receive the same fixed sequence of concurrent requests for repeated computation.

- Scalability indicates a system that can take full advantage of increasingly available parallel hardware. The commonly used multi-threading programming model used in SMP infrastructure has limits to its scalability because too much locking on shared resources can reduce scalability and concurrency.

- Composability is a desirable property, since composable modules are self-contained, and thus their behavior can be analyzed and tested in isolation. Composability guarantees that connecting the outputs of modules computing functions $f(x)$ and $g(x)$ to the inputs of a module computing function $h(x, y)$ will result in $h(f(x), g(x))$. Thus, components can be developed and tested individually, and later assembled into clustered hierarchy to create larger and more complicated systems.

- Boundedness is the ability of the modeled system to execute within finite amount of memory, which is practical for implementation in reality.

Architecture with these desirable properties can allow development of determinate, scalable systems built from composable components within bounded resources. Besides, each module can be written in the usual sequential manner where synchronization is implicitly controlled by communication primitives. Each module can thus be reused and does not need to concern about non-deterministic aspects of concurrent execution. The mechanism for achieving parallelization and distribution are also transparent. Several theoretical models have been developed to formally reason the message passing systems mathematically. Kahn Process Networks (KPN) [Kahn, 1974] is such typical formal model of concurrent computation in which a set of deterministic processes communicates via a series of unbounded first-in first-out (FIFO) queues. In the dataflow graph, the nodes are referred to actors. Each actor executes according to predefined rules that specify the action when tokens are available at its inputs. When executing, an actor consumes some number of input tokens and produces some number of output tokens.

There are basically two variations in how the message passing mechanism can behave. In the synchronous mechanism, a message sender will not proceed until the receiver has received the message. This provides synchronization points between concurrent components. However, it reduces concurrency because each point must wait for each other to proceed. In the asynchronous mechanism, by buffering messages and using asynchronous messaging, execution can overlap with communication.

Staged grid architecture applies an asynchronous message passing mechanism for concurrent model that is supposed to support scalability, composability and boundedness. The staged grid architecture can be abstracted as sequential nodes connected with channels. Each node represents a staged grid module with semantic behavior. Each node runs asynchronously and independently of other nodes, and its state is inaccessible to other nodes. Communication between nodes is only possible by sending and receiving discrete requests over unidirectional, reliable FIFO queue channels. A node may be connected to multiple channels, but each channel can be connected to exactly two nodes: one sender node and one receiver node.

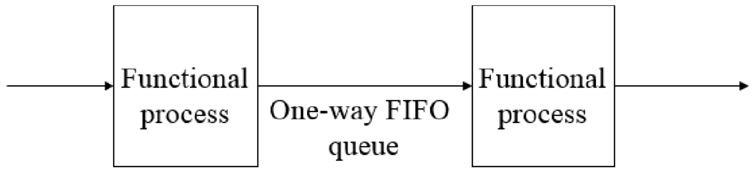


Figure 4.3: KPN Functional Process

Each transition of staged grid modules can be regarded as a functional process that maps an input operation sequence to an output sequence, as demonstrated in Figure 4.3. Each functional process has a pre-defined computing process, and these processes are networked together while complying with behavior execution rules at each port: blocking reads and non-blocking writes. That is, execution of a functional process is suspended when trying to consume operation from an empty channel queue, but is never suspended for producing output. A functional process cannot detect the presence of operation on an incoming queue, but attempt to read (and potentially block). Each queue is directed from one process to another as a one-direction FIFO. We now describe in detail the topologies and operational semantics of the staged grid architecture and its consequence, the most importance of which is deterministic execution.

The execution order consists of the order of the "*write*" and "*read*" operations on the data element. The $write(x)$ represents the writing or production of the data element x and $read(x)$ represents the reading or consumption of the data element x. We use the symbol $\prec$ to indicate the execution order.

- Since a data element must be written before it can be read, $write(x) \prec read(x)$ for every data element.
- The first-in-first-out (FIFO) property of the communication channels imposes

the restriction that $write(x_i) \prec write(x_j)$ if and only if $read(x_i) \prec read(x_j)$.

- Modular processes introduce additional restrictions. For example, if $W = f(U, V)$, then we have the following restrictions. The process reads from the two input sequence U and V. $read(u_i) \prec write(w_i), read(v_i) \prec write(w_i)$.

A sequential execution is a total ordering of write and read operations. We can compare any pair of operations. A parallel execution is a partial ordering of write and read operations. We may be able to compare some operations but will not be able to compare others. Any execution order must satisfy the restrictions imposed by the communication channels and the processes. The determinism within staged grid architecture means a unique history of each input operation sequence on channels despite the concurrent execution of modules. The deterministic property was verified through the introduction of the Kleene Least Fixed-point Theorem [Lee, 2011] of Kahn process network (KPN) [Kahn, 1974], according to which the history complies with the "least fixed point" of a set of mathematical equations. Limitations of determinism in the theoretical model are also discussed to figure out how to practically tackle these challenges when applying the model in the database system.

4.3.1 Monotonic and Continuous Function

We first demonstrate that the process of each staged grid module can be regarded as a monotonic and continuous function. In the above representation of staged grid architecture, all queues hold a sequence of operations.

A sequence is defined as a stream of data elements (or tokens), e.g. $X = \{x_0, x_1, x_2, ...\}$. In the context of a staged grid module, the contents of a sequence S represent the operations that have been inserted into a queue $\mathbf{S} = (\mathbf{e_1}, \mathbf{e_2}, \mathbf{e_3}...)$. Sequences are ordered in the prefix order.

Definition 6 *For a set of operations E and the set of finite and infinite sequences E^{**} of operations of E, the prefix order is a relation $\subseteq$ from E^{**} to E^{**} such that for any $s, s' \in E^{**}$, $s \subseteq s'$, if*

- *s is the empty sequence $\perp$ (the empty sequence $\perp$ is a prefix of all sequence).*

- *$\forall n \in N where$ $s(n)$ is defined, $s'(n)$ is defined and is equal to $s(n)$.*

For example, the sequence $X = [0]$ is a prefix of the sequence $Y = [0, 1]$, which is in turn a prefix of $Z = [0, 1, 2, ...]$.

The prefix order relation is a partial order on the set of all sequences satisfying the following properties. For sequences X, Y, Z

- Reflexivity: $X \subseteq X$
- Asymmetry: $X \subseteq Y \& Y \subseteq X \rightarrow X = Y$
- Transitivity: $X \subseteq Y \& Y \subseteq Z \rightarrow X \subseteq Z$

Tuples of sequences are also a partial order. For example, $(X_0, X_1) \subseteq (Y_0, Y_1)$ if and only if $X_0 \subseteq Y_0, X_1 \subseteq Y_1$.

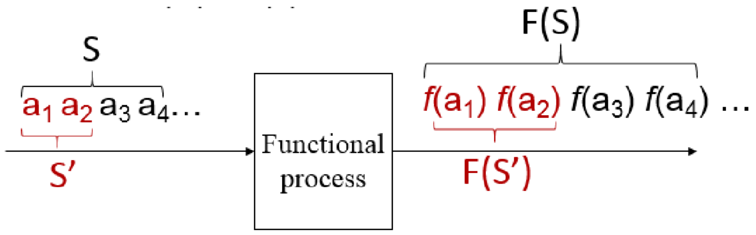


Figure 4.4: Monotonicity illustration

Since we applied sequential program for each staged grid module and enforced blocking reads rule, each module that takes **S** as an input will firstly read in a prefix sequence that equals to $\mathbf{S'}$, and the output will be exactly identical as what could be produced with $\mathbf{S'}$ being the input sequence; continuing reading left sequence of $\mathbf{S} - \mathbf{S'}$ will not affect previously produced output sequence. The output sequence in response to $\mathbf{S'}$ is also a prefix of the output sequence produced by **S**. Thus, the prefix order of sequence is preserved, as illustrated in Figure 4.4.

Each staged grid module transition can be regarded as a function $F : E^{**} \rightarrow E^{**}$ that maps the input sequence into the output sequence.

Guarantee 7 *$F : E^{**} \rightarrow E^{**}$ is a* ***monotonic*** *function preserving prefix order. If input sequences $S \in E^{**}$, $S' \in E^{**}$ are in prefix order which means S' is a prefix of S, $S' \subseteq S$, the output sequences $F(S'), F(S)$ preserve the prefix order $F(S') \subseteq F(S)$. That is, $S' \subseteq S \rightarrow F(S') \subseteq F(S)$.*

Since each staged grid module is coded with sequential program and enforces blocking reads rule, it takes S as an input will firstly read in a prefix sequence that equals to S', and the output will be exactly identical as what could be produced with S' being the input sequence. Continuing reading left sequence of $S - S'$ will not affect previously produced output sequence. The output sequence in response to S' is also a prefix of the output sequence produced by S. Thus, the prefix order of sequence is preserved.

Function with monotonicity property implies that it is safe to evaluate the function with partial information about its input. Given only a prefix of what the input will eventually be, the function can be evaluated using only the prefix, and we will get a prefix about what the output will eventually be when the input is complete.

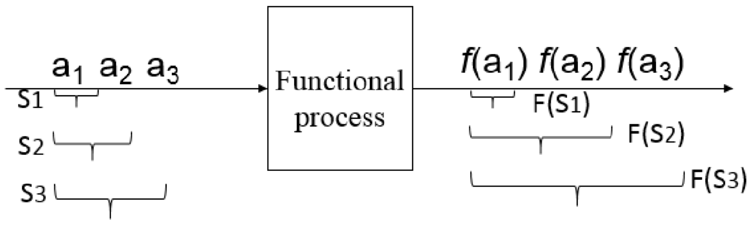


Figure 4.5: Increasing Chain of Sequences

The entire history of operations that have been inserted into a queue as execution progresses can be represented as an increasing chain of sequences $\vec{S} = (S_0, S_1, S_2...)$ where $S_0 \subseteq S_1 \subseteq S_2...$, as shown in Figure 4.5. Each sequence element of $\vec{S}$ contains the operation history of the channel at some certain point in the execution.

Definition 8 *The least upper bound of the chain $\vec{S} = (S_0, S_1, S_2...)$, denoted by $\vee(\vec{S})$, is the shortest sequence such that $\forall S \in \vec{S}, S \subseteq \vee(\vec{S})$. For any chain of sequences $\vec{S}$, the least upper bound $\vee(\vec{S})$ exists and is unique: $\vee(\vec{S}) \equiv \lim_{n\to\infty} S_n$.*

Guarantee 9 *$F : E^{**} \to E^{**}$ is a **continuous** function on an increasing chain $\vec{S}$=(S_0,S_1,S_2...), where $S_0 \subseteq S_1 \subseteq S_2$... That is, $F(\vee(\vec{S})) = \vee(F(\vec{S}))$.*

Continuous function ensures that a function does not "wait forever" before producing output. Suppose that the chain $\vec{S} = \{S_0, S_1, S_2, ...\}$ represents a sequence of partially constructed inputs to a process. Then $\vee(\vec{S})$ represents the eventually complete input. The set $F(\vec{S}) = \{F(S_0), F(S_1), F(S_2), ...\}$ represents the partially constructed outputs, given partially constructed inputs. Then continuity requires that $\vee(F(\vec{S}))$ be equal to the eventually complete output $F(\vee(\vec{S}))$.

The least upper bound of the chain $\vee(\vec{S})$ is the shortest sequence such that all sequences in the chain are a prefix of $\vee(\vec{S})$. Since the chain $\vec{S}$ represents a sequence of partial inputs to each module, $\vee(\vec{S})$ is the eventual complete input. The unblocking-write behavior rule enforces that each module does not halt before producing the corresponding complete output. Using the least upper bound of a sequence as input, the output is also the least upper bound.

Since the continuous and monotonic function has the property of compositionality, thus the whole staged grid architecture can be regarded as a continuous and monotonic function composed of the functions defined by each individual staged module function over increasing sequence chains.

4.3.2 Least Fixed Point

Since the continuous and monotonic function has the property of compositionality, thus the whole staged grid architecture can be regarded as a continuous and monotonic function composed of the functions defined by each individual staged module function over increasing sequence chains. The unique execution history of the architecture can be reduced to finding the fixed point of a function. Given a function $f : X \to X$. If there is a value $x \in \vec{X}$ such that $f(x) = x$, that value is called a fixed point.

The Kleene Least Fixed-point theorem assures the existence and uniqueness of such a least fixed point for monotonic and continuous function [Lee, 2011]. Existence and uniqueness of fixed point is essential for the semantics of executions.

Theorem 10 ***Kleene Least Fixed-point theorem*** *[Lee, 2011]: For any monotonic function $F : \vec{X} \to \vec{X}$, where $\vec{X}$ is a chain. Suppose a sequence $\perp$ is the prefix of all sequences in the chain $\vec{X}$, i.e. $\forall X \in \vec{X}, \perp \subseteq \vec{X}$. Let $C = F^n(\perp) : n \in N$. Then if $\vee X = F(\vee X)$, $\vee X$ is the least fixed point of F. If F is also continuous, then $\vee X = F(\vee X)$.*

Let α be any fixed point, i.e. $f(\alpha) = \alpha$. If $\vee X \subseteq \alpha$, $\vee X$ is the least fixed point. Since $\perp \subseteq \alpha$ and F is monotonic, $F(\perp) \subseteq F(\alpha) = \alpha$. Recursively using the monotonic property, this implies that $F(F(\perp)) \subseteq F(F(\alpha)) = F(\alpha) = \alpha, \forall n \in N$,

$F^n(\perp) \subseteq F^n(\alpha) = \alpha$. Also $\perp \subseteq F(\perp) \subseteq F(F(\perp)) \subseteq F(F(F(\perp)))...\forall n \in N, F^n(\perp) \subseteq F^{n+1}(\perp)$. Let $C = F^n(\perp) : n \in N.F(C) \cup \perp = C$. Moreover, $\vee(F(C) \cup \perp) = \vee(F(C))$ since an additional $\perp$ in a chain will not change its least upper bound. Thus we have $\vee F(C) = \vee(C)$, and $\vee(C)$ is the least fixed point.

Guarantee 11 *$F : E^{**} \to E^{**}$ holds a unique least fixed point obtained by recursively iterating the function F on the least prefix element $\perp$, generating an increasing chain: $\vec{S} = (\perp, F(\perp), F(F(\perp))...F^n(\perp)...)$.*

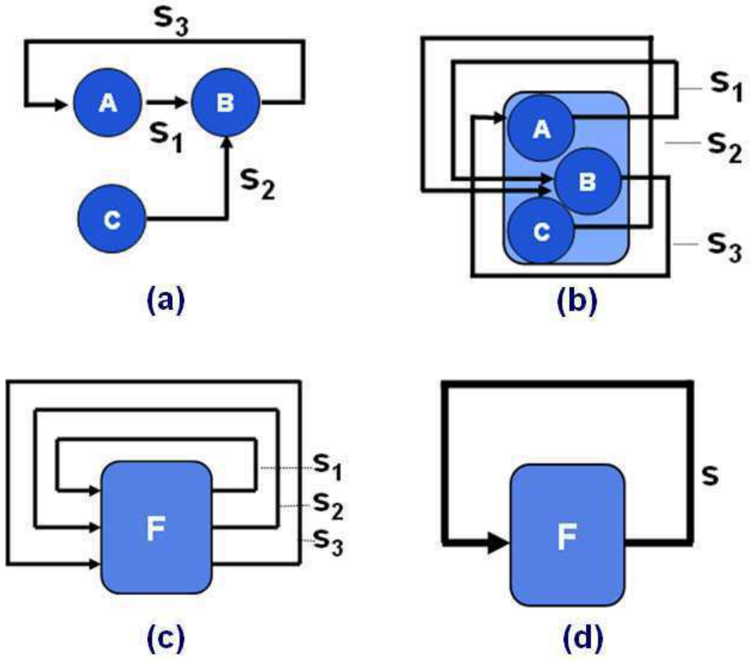


Figure 4.6: Execution transformation to a fixed point of a function

Figure 4.6 demonstrates Kleene Least fixed-point theorem in a simple example. Given three modules with three channels, modules can be re-drawn into one rectangle forming a combined function. Then three channels become undifferentiated and can be merged into one channel. Therefore, the execution is reduced to a fixed point of a function.

Kleene Least fixed-point theorem provides the mathematically provable property of determinism regardless of the concurrent execution order of the processes in the program. The sequence of the history of operations on all of the queues depends only on the solution to the fixed point equation.

4.3.3 Effective Execusion

The previous section shows that the history of tokens on all of the queues depends only on the solution to the fixed point equation. However, the assumption about the termination of each modular program is undecidable infinite time, as is boundedness of the queues. Boundedness is related to the number of unconsumed operations in the queues during the execution.

The interpretation of the infinite sequence is that the system does not terminate, which can continue to execute for as long as it is expected to execute. The sequences that have been produced at any point during the execution are mere approximations. To make the approximation appropriate, the execution requires complying with the principle as following.

Definition 12 *For a module process, which is **fair** with respect to its inputs and outputs, the ability to produce an output sequence or read an input sequence will eventually be allowed to realize.*

A fair process cannot indefinitely neglect any of its inputs or outputs. A fair processes will allow the sequence to proceed and eventually converge to the least fixed point [Geilen and Basten, 2003].

Theorem 13 ***Kahn principle****: An execution is effective if each module process involved is fair. Any two effective executions produce the same sequences of messages, matching the least fixed point within the semantics [Kahn, 1974; Lynch and Stark, 1989].*

The Kahn principle states that fair executions produce sequences that match the least fixed point. Every effective execution is finite in practice and converge in the limit to the least fixed point [Lee, 2011]. To achieve effective executions, we implemented reliable FIFO queue established on TCP/IP socket for channels. The reliability guarantees every operation produced by a source process will eventually be read by the destination module process.

For an important class of models complying with Kahn principle, termination is a property of the program and does not depend on the execution order. However, the number of data elements buffered on the communication channels during execution does depend on the execution order and is not completely determined by the programs definition. Thus, the mechanism is needed to guarantee the number of data elements buffered on the communication channels remains bounded for all possible execution orders.

The unbounded queues assumption makes the implementation of the system infeasible in bounded memory, since the queues will grow to be of infinite length, and are therefore impossible to run in a limited memory space. To make the program able to be implemented within infinite memory, the number of unconsumed operations in queues should be finite for bounded programs. Though the "*write*" semantics is defined to be unblocked, however, the "*write*" has to be blocked if the output channel is full. With at least one module being blocked on "*write*", a subset of modules in the system will be blocked. This kind of block is artificial because it would not occur with infinite channel capacities.

By requiring all processes to be fair, we need to make sure that any full queues are eventually and periodically read from, and any such blocked producers will eventually make progress. As long as modules that are blocked on full queues can eventually make progress, the system will eventually proceed and converge to the least fixed point.

When execution repeatedly blocks due to full channels, we increase the size of at least one channel capacity on which a module process is "*write*" blocked. The system continues to execute unbounded as long as the system resources are not exhausted, which eventually will be blocked when all module processes are blocked by reading from empty channels and no modules are blocked by writing to full channels. It is proved that this strategy delivers an effective execution that transforms a complete, unbounded execution to a partial bounded execution [Lee and Parks, 1995; Geilen and Basten, 2003].

Guarantee 14 *The staged grid architecture yields deterministic effective execution converging to the least fixed point with blocking-reads & unblocking-writes behavior semantics, reliable FIFO queue bounded channels and artificial blocking resolution strategy.*

4.4 Chapter Summary

Inspired by the principle behind the staged event driven architecture and MapReduce framework, in this chapter, we have introduced a scalable staged grid architecture for NewSQL database management systems, which can be elastically deployed on the shared-nothing infrastructure.

We define a highly scalable staged grid database architecture, and provide its implementation techniques in details, based on which a NewSQL database system Rubato DB is implemented. We disintegrate and reassemble the traditional single server database system design into encapsulated staged modules. Then independent staged modules deployed on networked computing nodes are orchestrated based on communication to work as one single database instance.

To verify the proposed architecture, we prove the behavior of the systems on the architecture is deterministic for applications. Determinism in the staged grid database architecture context means that the program can always generate the same procedure despite the concurrent execution of the nodes deployed on the shared-nothing infrastructure.

Chapter 5

Formula Protocol for Concurrency Control

"Of course there is no formula for success, except perhaps an unconditional acceptance of life, and what it brings."

- Arthur Rubinstein.

Scaling out while processing transactions efficiently is an important requirement for NewSQL database systems. We analyzed multiple popular mechanisms in current DBMSs to distill some feasibility and principles for building systems that scale out to clusters of commodity servers while efficiently executing transactions. In this chapter, we first highlight these designs and then present the *Formula Protocol for Concurrency* (FPC), the concurrency control protocol used in Rubato DB; and then outline our schema for the implementation of the protocol.

5.1 Formula Protocol Overview

One commonly used implementation to provide serializability in large-scale database systems, such as Megastore [Baker *et al.*, 2011], Spinnaker [Rao *et al.*, 2011], etc., is based on distributed two-phase locking with two-phase commit protocol.

Using a locking-based concurrency control mechanism, update transactions are not permitted to write a data object which has been read by a read transaction and thus has to wait until the potential long-running read transaction completes. All of a transaction's locks must be held for the whole duration of the commit phase. Since multiple network round-trips are needed for commit, the extra time that locks are held can considerably reduce the overall transactional efficiency [Thomson *et al.*, 2012]. Besides, the deadlock detection and resolution may further prohibit the concurrency and scalability [Larson *et al.*, 2011].

Some systems (e.g. Azure [Campbell *et al.*, 2010], Sinfonia [Aguilera *et al.*, 2009] and H-Store [Jones *et al.*, 2010; Kallman *et al.*, 2008]) attempt to reduce the locking overhead by constraining the scope of transactions whose accesses are limited to a small subset of database. Transactions that can be executed in parallel to completion without requiring communication with others are optimized, but other transactions are aborted or executed with coarse-grained locks on each partition. Despite of the high performance and scalability induced by restricted transactional scope, the concurrency and applicability can be hindered for generalized workload [Baker *et al.*, 2011; Kallman *et al.*, 2008; Thomson *et al.*, 2012]. Another simplified variation is to write transactions as stored procedures [Stonebraker *et al.*, 2007] which can be executed sequentially within different partitions to take advantage of parallelism. Calvin [Thomson *et al.*, 2012] introduced deterministic ordered locking. By using a pre-ordered agreement for acquiring locks in the presence of distributed transactions, distributed commit protocols can be eschewed. However, stored procedures and ordered locking require extra efforts to analyze knowledge of all transactions in advance, which can be time-consuming and error-prone for complicated workloads.

Multiversion Concurrency Control (MVCC) provides a better solution, in which each update creates a new version of that data object rather than update the data object in place. Concurrent read can access the old version of data while the update transaction can proceeds without waiting. Without using locking mechanism, reads and writes are not necessarily block each other.

Most typical systems using MVCC, such as Microsoft Azure [Campbell *et al.*, 2010] and SAP HANA [Sikka *et al.*, 2012], only guarantee a weaker isolation level than serializability, such as read committed or snapshot isolation. Although those weak guarantees offers good isolation level, some non-serializable schedules are still permitted. The known solution to avoid non-serializable schedules in weak isolation level is to keep track of the entire read/write set of transactions, which tends to create an expensive overhead [Jorwekar *et al.*, 2007; Cahill *et al.*, 2009].

The formula protocol for concurrency (FPC) used in Rubato DB is a novel implementation of the classical Multiversion Timestamp Concurrency Control Protocol [Thomas, 1984] with new features: **logical transformation formula caching** and **dynamic timestamp ordering**, for achieving serializability with scalability.

The **logical transformation formula caching** approach has the advantages over storing actual multiple versions of data items mainly in the following cases:

(a) For all non-key updates, storing multiple versions need to maintain physical copies of numerous rows [Manassiev *et al.*, 2006], but the formula-based approach still uses one single formula. This will significantly reduce overhead of multiple versions of all update data items.

Example 15 *Consider a database containing one table "Employees" with the initial instances as shown in Table 5.1, and a request from one transaction:*
T_{10}*: UPDATE employees SET* $salary = salary * 1.1$ *WHERE* $d\_id = 20$*;*

Table 5.1: Table Employees Instance

e_id	**d_id**	**salary**
1001	10	90
2000	20	100
2001	20	100
...	...	...
2999	20	100
3000	30	110

Since there exist 1000 records in the table satisfied the condition $d\_id = 20$, in the traditional MVCC, new versions of data (version= T_{10}, $d_{id} = 20$, $salary = 110$) require to be generated for all of the 1000 records ($e_{id} = 2000, ..., 2999$).
However, within the FPC, one single formula $update(T_{10}, d\_id = 20 \rightarrow salary = salary * 1.1)$ is enough for all 1000 records.

(b) Formula enables us to use conflict-free operations such as increment/decrement, instead of updates, which is much easier than if otherwise, in terms of much less conflict potential.

Example 16 *Consider Table 5.2 and two requests from different transactions:*
T_{20}*: UPDATE employees SET* $salary = salary + 20$ *WHERE* $d\_id = 10$*;*
T_{30}*: UPDATE employees SET* $salary = salary - 30$ *WHERE* $d\_id = 10$*;*

Since T_{20}, T_{30} both update the same data item in the table, they are considered to be conflict. In the traditional MVCC, a new version of data copy V_{20} (version= 20, $e_{id} = 1001$, $d_{id} = 10$, $salary = 110$) will be created according the update request in T_{20}. And another new version of data copy V_{30}(version= 30, $e_{id} = 1001$, $d_{id} = 10$, $salary = 60$) is also generated for T_{30}.
Only one of V_{20} and V_{30} can be the final result due to the conflict between T_{20} and T_{30}. Usually, rules such as "first commit writer wins" are commonly applied [Fekete *et al.*, 2005; Cahill *et al.*, 2009]. For instance, once T_{30} commits before T_{20}, the data version V_{20} of T_{20} will be discarded and T_{20} needs to be rolled back and retried.

However, actually the execution order of those two operations does not affect the final result, thus such operations should be considered conflict-free, even though they write on the same data item. Within FPC, if two formulas are generated from conflict-free operations, they are considered commutative and FPC does not need to unnecessarily abort any of the transactions.

(c) The implementation of multiversion uses the fine-grained page as the minimal unit in the memory [Blott and Korth, 2002; Brantner *et al.*, 2008]. The page size can affect the latency greatly since the full bandwidth can only be achieved if data is flushed in pages of relatively large size (e.g. 4KB, 8KB) [Brantner *et al.*, 2008]. To save page space, multiple versions of different data are clustered into pages, and complex and error-prone mechanism is required to ensure the update operation does not overwrite each other even though they access different data [Blott and Korth, 2002; Brantner *et al.*, 2008].

Example 17 *Consider transactions* T_{10}*,* T_{20} *and two pages* P_1*,* P_2*.*

Assume the original copy of employees lives on the page P_1*.* T_{10} *reads data from* P_1 *and creates a new version* V_{10} *on the page* P_2*. Similarly,* T_{20} *reads data from*

P_1 and creates a new version V_{20} on the page P_2.

The two transactions T_{10} and T_{20} have no overlapping read sets and generate different versions of data. However, since the order in which T_{10} and T_{20} arrive at the scheduler is same to the order of their assigned versions, normally T_{10} locks the page P_2 and creates version V_{10} on P_2. Subsequently, T_{20} may need to wait until T_{10} commits and releases its lock on page P_2 in order to be able to create its required version V_{20} on P_2. This wait is due to a write-write conflict on the page P_2 upon creating different versions, rather than a conflict on the actual same data access. Note that such wait duration would normally be longer or involving more pages if T_{10} has updated a large set of data. Instead, our FPC modification process can minimize the conflict wait time.

In addition, since the size of data version V_{10}, V_{20} varies, extra efforts are required for the alignment in the page.

The manipulation on the unit of formula can reduce the complexity of storing multiple versions on the page-level. FPC is also suitable for both numeric values and string values.

By storing the update formulas locally associated with the data items, it not only significantly reduces overhead of storing multiple updates data items, but also enables us to achieve serializability over distributed servers efficiently. For most transactions which are read-only, FPC can also take advantage of the transaction ordering perspective.

The **dynamic timestamp ordering** is used to achieve:

1. avoiding unnecessary blocking or waiting in order to increase the degree of concurrency.

2. clearing the formulas as early as possible, similarly to database management systems using the multiversion timestamp protocol that delete any version of updated data item as soon as it is not needed.

In the conventional multiversion timestamp protocol, the commit order of operations conforms with the timestamp initially assigned to each transaction. This mechanism is considered to be **static**. However, the FPC respects the initial timestamp ordering while permitting an equivalent schedule that differs from the static timestamp ordering, as long as it ensures serializability. The timestamp ordering of all the transactions may be altered to allow a transaction with older (smaller) timestamp to read the data item updated by a later (larger) transaction on condition that the serializability is respected.

The following example demonstrates the two distinct features of the FPC.

Example 18 *Consider a database containing one table* **employees***, with the initial instances as shown in Table 5.2.*

Table 5.2: Employees Instance

e_id	**d_id**	**salary**
1	10	90
2	20	100
3	20	80

Let A, B, C denote three values in one data item[1]*. We use "W", "R" to denote "write" and "read" operation respectively, e.g. $W(B = B + 10)$ means update B by increasing the value of B by 10; $R(B)$ means read the value of B. Consider the schedule in Table 5.3 where T_{10}, T_{20}, T_{30} are three transactions with subscripts as their timestamp:*

Table 5.3: Transactions Schedule

	T_{10}	T_{20}	T_{30}	**A**	**B**	**C**
t_1				90	100	80
t_2	W(B=B×1.1)			90	100	80
t_3			W(B=B+10)	90	100	80
t_4			W(C=C+10)	90	100	80
t_5			commit	90	110	90
t_6		R(B=121)				
t_7	commit			90	121	90
t_8		commit		90	121	90

[1]For simplicity, we consider only one data item but all the discussions are valid for a set of data items.

In this schedule, the updates issued by T_{10}, T_{30} at t_2, t_3, t_4 will be executed by caching formulas rather than multiple versions of replica. When T_{30} requests to commit at t_5, all formulas issued by T_{30} (i.e., $W(B = B + 10)$, $W(C = C + 10)$) will be force-written and eliminated from the memory. After T_{30} commits, the disk values of B and C are updated to 110 and 90 respectively. Further, when T_{20} issues $R(B)$ at t_6, it will first read the value of B from disk as 110, and then the update formula $W(B = B * 1.1)$ issued by T_{10} earlier will be applied. Hence, T_{20} reads the value of B as 121. This schedule that is not allowed in the classical multiversion timestamp protocol, is indeed serializable and its equivalent serial schedule is T_{30}, T_{10}, T_{20}. The schedule does not comply with the intial timestamp ordering, but it does permit an equivalent serial schedule.

5.2 Timestamp-based Formula Protocol

As with the timestamp protocol, each transaction under FPC is assigned a unique timestamp, $TS(T)$, when it is initiated in the socket monitor stage on one dedicated server. The FPC guarantees the existence of an equivalent serial schedule in which transactions are ordered by their timestamps (subject to dynamic timestamp ordering). The FPC stores with each data item, x, on relevant node, the following pieces of information:

- $lrt(x, Ni)$: the largest timestamp of active (not committed) transactions that have read x on the node N_i;
- $list(x, N_i)$: the list of update formulas of the form: $uf(x, T_{u1}, N_i), \ldots, uf(x, T_{un}, N_i)$, where $uf(x, T_j, N_i)$ represents an update formula on x by transaction T_j on Node N_i, and $TS(T_{u1}) \leq TS(T_{u2}) \leq \cdots \leq TS(T_{un})$.

If such an active transaction does not exist, $lrt(x, Ni)$, and $list(x, N_i)$ are set to 0 and $\emptyset$ respectively.

Read/Write Operation

When a transaction, T_1, makes a request to read x on node N_i, $read(x, T_1, N_i)$ will first retrieve the value of x from disk on N_i and then update the retrieved value using stored update formulas in $list(x, N_i)$ if needed. More specifically,

- **R1.** T_1 read x on N_i. Let $v_0(x)$ be the disk value of x on N_i, and

  $uf(x, T_{u1}, N_i), \ldots, uf(x, T_{um}, N_i)$ be the list of update formulas in $list(x, N_i)$ such that

  (1) $TS(T_{um}) \leq TS(T_1)$ and

  (2) $TS(T_{u(m+1)}) > TS(T_1)$.

  Then

  $v_1(x)$ be the value obtained by applying $uf(x, T_{u1}, N_i)$ on $v_0(x)$,

  $v_2(x)$ be the value obtained by applying $uf(x, T_{u2}, N_i)$ on $v_1(x)$, and $v_{um}(x)$ be the value obtained by applying $uf(x, T_{um}, N_i)$ on $v_{u(m-1)}(x)$.

  $v_{um}(x)$ is the value to be retrieved by $read(x, T_1, N_i)$.

  If $TS(T_1) > lrt(x, N_i)$, $TS(T_1)$ is assigned to $lrt(x, N_i)$.

  To facilitate the cascading rollbacks and $wait\_for\_commits$, the FPC protocol also records a fact:

  $read\_by(T_{uk}, x, T_1)$ for $1 \leq k \leq m$,

  indicating the value x updated by T_{uk} is read by T_1.

Obviously, T_1 retrieves the value that is obtained by sequentially applying all update formulas on x issued by transactions with older timestamps, and if $TS(T_1) < swt(x, N_i)$, where $swt(x, N_i)$ is the smallest timestamp of a transaction contained in $list(x, N_i)$, it retrieves the disk value of x directly. This is the same as the multiversion timestamp protocol, except that the update formulas are used instead of actual multiple versions of data items.

When a transaction, T_1 with timestamp $TS(T_1)$, makes a request to write x on node N_i, i.e., $write(x, T_1, N_i)$, the FPC performs the following action:

- **W1.** If $TS(T_1) < lrt(x, Ni)$, there must exist another transaction T_2, which should follow T_1 according to the equivalent serial order on the timestamp, has read the value of x before.

  Thus T_1 is too old to write x, and must be aborted.

This may also trigger cascading rollbacks as in **K1**.

- **W2.** If $lrt(x, Ni) = 0$ or $TS(T_1) = lrt(x, N_i)$, the write request is processed by simply adding a new update formula

  $uf(x, T_1, N_i)$ into $list(x, N_i)$.

- **W3.** If $TS(T_1) > lrt(x, N_i) > 0$, the write request is processed by simply adding a new update formula $uf(x, T_1, N_i)$ into $list(x, N_i)$.

  Further, there must exist another transaction T_2 such that

  (a) $TS(T_2) < TS(T_1)$, and

  (b) T_2 has read the value of x before.

  The FPC protocol also records a fact $read\_b4(T_2, x, T_1)$.

In **W3**, $read\_b4(T_2, x, T_1)$ is recorded to facilitate the dynamic timestamp ordering. Assume that $TS(T_2) < TS(T_1)$, and T_1 issues commit before T_2 does. The dynamic timestamp ordering allows T_1 to commit, but update formulas of T_1 will be retained in the memory rather than cleared if there exists any $read\_b4(T_2, x, T_1)$ stored in the system. Only when all the stored facts $read\_b4(T_i, x, T_1)$ for T_1 are removed as in **C2**, **K1**, update formulas of T_1 can be cleared and force-written to disks.

Commit/Rollback Protocol

To facilitate committing a transaction that has accessed data items distributed over different nodes, the FPC maintains a list $PN(T)$ of participating nodes for each active transaction T. That is,

$$PN(T) = \{N_i \mid T \text{ reads } x \text{ on } N_i \text{ or } T \text{ writes } x \text{ on } N_i\}.$$

When a transaction, T_1, makes a request to commit, the FPC performs a variation of two-phase commit protocol. In the 1st-phase, pre-commit(T_1,N_i) pre-checks if N_i is ready to commit for each $N_i \in PN(T_1)$. In the 2nd-phase, a consentaneous action will be taken based on the response from each node. There are two kinds of approaches to present the dynamic timestamp ordering: one is pessimistic that forces transactions to $wait\_for\_commit$ for $read\_b4$ facts; the other is optimistic that allows transactions to commit immediately in presence of $read\_b4$ facts, but retain the formulas without being cleared until all $read\_b4$ facts are removed later. The pessimistic approach is optimal for read intensive workloads where $read\_b4$ facts are rare; while the optimistic approach is optimal for write intensive workloads where $read\_b4$ facts are frequent.

- **P1. (pessimistic)** pre-commit(T_1,N_i)

    1. If there exists $read\_by(T_2, x, T_1)$ or $read\_b4(T_2, x, T_1)$ on N_i, suspend T_1 with $wait\_for\_commit(T_1, T_2)$.
    2. Otherwise, return T_1 is ready to commit on N_i.

- **P2. (optimistic)** pre-commit(T_1,N_i)

    1. If there exists $read\_by(T_2, x, T_1)$ on N_i, suspend T_1 with $wait\_for\_commit(T_1, T_2)$.
    2. Otherwise, return T_1 is ready to commit on N_i.

The FPC waits until it receives the ready-to-commit message from all nodes in $PN(T_1)$, then performs commit(T_1,N_i) on all nodes $N_i \in PN(T_1)$.

- **C1. (pessimistic)** commit(T_1,N_i), i.e., T_1 commits on the node N_i: Force-write all update formulas issued by T_1 on N_i, and remove all stored facts, such as $read\_by$, $read\_b4$, and uf involving T_1.

- **C2. (optimistic)** commit(T_1,N_i) i.e., T_1 commits on the node N_i: The formulas issued by T_1 are cleared and force-written to database disks only if there exists no $read\_b4(T_{rb}, x, T_1)$ fact; otherwise formulas are reserved in the memory. When T_{rb} commits or rollbacks and $read\_b4(T_{rb}, x, T_1)$ is the only $read\_b4$ stored fact for T_1, force-write and clear all update formulas issued by T_1 on N_i in **W3**.

After T_1 commits in all participating nodes in $PN(T_1)$, the FPC wakes up all waiting transactions T_w to resume pre-commit(T_w,N_i), if $wait\_for\_commit(T_1, T_w)$ recorded in **P1**(1) or **P2**(1). When a transaction T_1 makes a request to rollback, or is forced to rollback as in **W1**, or times out to receive any response from N_i due to network failure, the FPC performs rollback(T_1,N_i) on all $N_i \in PN(T_1)$.

- **K1.** rollback(T_1,N_i). Send cascading rollbacks to all T_r if $read\_by(T_r, x, T_1)$ is recorded; and remove all stored facts, such as $read\_by$, $read\_b4$, and uf involving T_1 on N_i. If there exists $read\_b4(T_1, x, T_{cb})$ and it is the only $read\_b4$ stored fact for T_{cb}, force-write and clear all update formulas issued by T_{cb} on N_i in **W3**.

After T_1 rollbacks in all participating nodes in $PN(T_1)$, the FPC wakes up all waiting transactions T_w to be rolled back. if $wait\_for\_commit(T_1, T_w)$ is recorded in **P1**(1) or **P2**(1). As indicated after **C1**, **C2** and **K1**, when T_1 terminates (commit or rollback), all transactions that are waiting for T_1 resume and restart to pre-commit or cascading rollback. The transition for transaction states is demonstrated in Figure 5.1.

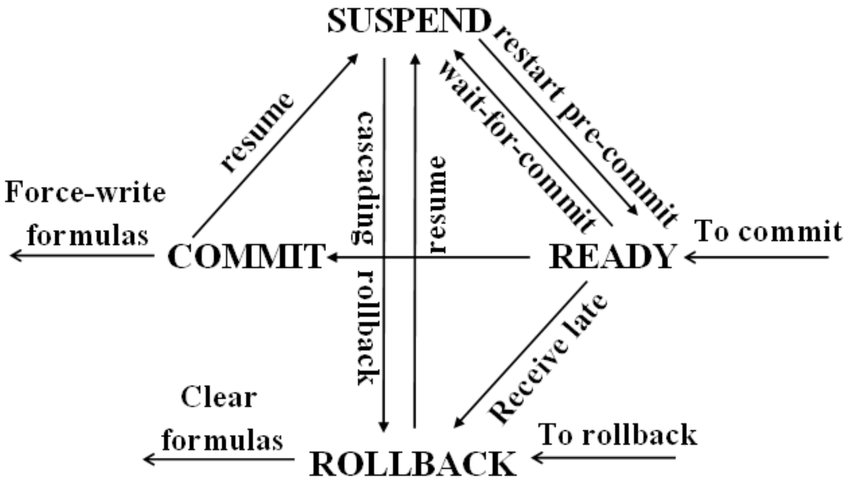


Figure 5.1: Transaction states Transitions

5.3 FPC Serializability Guarantee

Since FPC is a novel implementation of the multiversion timestamp protocol [Thomas, 1984], we can prove that the FPC guarantees serializability and stomicity sticking to the conventional notations in the multi-version protocol. As a matter of fact, its proof follows from the following facts:

(a) The orders of all conflict pairs of operations are compatible with the timestamp ordering of all the involving transactions.

(b) The dynamic ordering rearranges a transaction T to be effectively a smaller timestamp only if T has no operations that are conflicting to any operations of T_i such that $TS(T_i) < TS(T)$.

(c) Update transactions commit corresponding to timestamp ordering. If T_1 and T_2 are update transactions that commit, then if $TS(T_1) < TS(T_2)$ (after dynamic ordering if possible), then T_1 commits before T_2.

(d) No transaction commits in a state where it has read uncommitted data. That is, transactions that issue commit will wait for any uncommitted update transactions from which they read.

(e) Any transaction that reads data written by an aborted transaction itself aborts.

We use two abstractions in the FPC, timestamps (versions) and conflict dependencies, to reason about serializability isolation. Since FPC is a variation of multi-version implementation, we still stick to the conventional notations in multi-version protocol for the serializability guarantee proof. We assume $w_i(x_i)$ denotes a new version x_i issued by the write operation w_i in transaction T_i; and $r_i(x_j)$ denotes the read operation op_i in transaction T_i on the data version x_j. The multi-version history H_x defines a total order $\prec_x$ for each data object x. The total version order $\prec$ is the union of the $\prec_x$ in H_x [Thomas, 1984].

Given a multiversion history H and a total version order $\prec$, a direct dependency graph can be constructed by setting a vertex for each transaction T_i in H, and a direct edge $T_i \rightarrow T_j$ if one the following conditions hold:

(a) *write-read* dependency: $T_i \rightarrow_{wr} T_j$, if $w_i(x_i)$ in T_i produces a version of data object x_i, and $r_j(x_i)$ in T_j reads the version of data object x_i.

(b) *read-write* dependency: $T_i \rightarrow_{rw} T_j$, if $r_i(x_k)$ in T_i reads a version of data object x_k, and $w_j(x_j)$ in T_j produces a later version of data object x_j.

(c) *write-write* dependency: $T_i \rightarrow_{ww} T_j$, if $w_i(x_i)$ in T_i produces a version of data object x_i, and $w_j(x_j)$ in T_j produces a later version of data object x_j.

Guarantee 19 *The history H generated by FPC can guarantee serializability as its corresponding direct dependency graph does not contain any oriented cycle.*

Proof. Our proof is based on establishing a mapping between each vertex in the direct graph and its dynamic timestamp order. We prove the graph to be acyclic by illustrating that for each edge $T_i \rightarrow T_j$ in the graph, FPC can guarantee that $TS(op_i) < TS(op_j)$.

- In the case (a), $T_i \rightarrow_{wr} T_j$ means that r_j in T_j has read a version of data object produced by w_i in T_i.

  1. If T_i has a smaller timestamp than T_j, according to the writing rule **W1**, w_i arrives too late so that T_i must be aborted and trigger rollback as in **K1**.
  2. Otherwise, a *read_by* fact according to the reading rule **R1** is generated and regulates $TS(w_i) < TS(r_j)$ in **P1** or **P2**.

- In the case (b), $T_i \rightarrow_{rw} T_j$ means that r_i reads a version of data object before w_j in T_j produces a later version. A stored fact *read_b*4(T_i, T_j) will be generated and stored according to the writing rule **W3**.

  The *read_b*4 fact in **W3** guarantees that T_j with a larger timestamp can not be committed before T_i with a smaller timestamp based on **P1, C1** or **P2, C2**. Therefore, we can also have $TS(r_i) < TS(w_j)$.

- In the case (c), $T_i \rightarrow_{ww} T_j$ means that T_j produces a later version of data object overwriting the version created by T_i.

  1. If there exists any transaction T_k such that a read operation r_k in T_k reads the version newer than x_i while older than x_j, then we can have $T_i \rightarrow_{wr} T_k$ and $T_k \rightarrow_{rw} T_j$.

     Based on previous case (a) and case (b), $TS(T_k) < TS(T_j)$ and $TS(T_k) < TS(T_j)$ will be held accordingly.

     Therefore, we can have $TS(T_i) < TS(T_k) < TS(T_j)$.
  2. Otherwise, either T_i or T_j can commit at first in **C1** or **C2**, since we only regulates read_b4, read_by in **P1** and **P2**, which will not suspend T_i or T_j. And thus the dynamic timestamp ordering may be considered as alternating the timestamps to guarantee $TS(w_i) < TS(w_j)$.

■

5.4 Implementation of FPC

Rubato DB uses Berkeley DB[2] with B-tree indexes for basic data operations (e.g. *put*, *get*, *insert*, *delete*). All table partitions and their secondary index files if any are stored on the local disk as Berkeley DB files. The basic operations of the FPC are implemented as a layer, called Formula DB (FDB) on top of Berkeley DB. Formula DB is a thread-free package compatible with Berkeley DB, such that all disk accesses are through Formula DB[3]. In addition to all Berkeley DB operations, Formula DB also supports the following operations and functionalities:

(a) FDB$\rightarrow$ pre-commit(T_1): to perform the pre-commit operation on FDB, as specified in **P1** and **P2**,

(b) FDB$\rightarrow$ commit(T_1): to perform the commit operation on FDB, as specified in **C1** and **C2**.

(c) FDB$\rightarrow$ rollback(T_1): to perform the rollback operation on FDB, as specified in **K1**.

(d) FDB$\rightarrow$ update(T_1, N, F, W): to update the Nth column of the FDB according to the formula F for all records satisfying the boolean value expression W.

Formula DB uses (a), (b), (c) to achieve serializability over distributed nodes; and uses (d) to store logic formulas instead of multiversions of updated data items for FPC. Formula DB implements a fine granularity of control on the manipulation of data at row level or even finer by using formula unification. As shown in Figure 5.2, an internal main memory Formula DB is used to store logic formulas of all the updates. Each update operation is stored as one formula. Selection or read operation is evaluated against both formulas and disk. Update or write operation is done only after the transaction is committed. All associated formulas are removed from the main memory when the related transaction's state is cleared. The implementation of Formula DB is decentralized in that every node contains a transaction stage to initialize the two-phase commit/rollback.

[2]The transaction support of Berkeley DB itself is turned off.

[3]We set the buffer size of Formula DB same as Berkeley DB that is fine-tuned based on the data size.

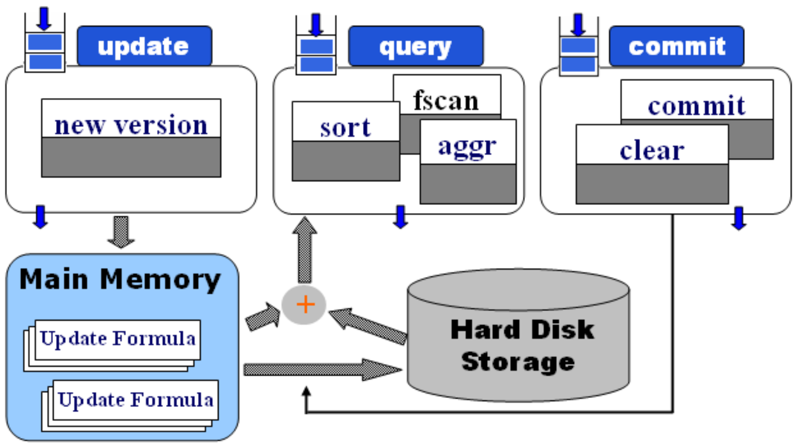


Figure 5.2: Query/Update engine module

FPC operates in a distributed environment, which needs to deal with partial failure and network partitions. Partition independence ensures that failed clients do not cause other clients to fail. This provides network partition tolerance as long as clients can access relevant working partitions.

FPC uses a two-phase commit protocol, which will always complete the transaction except when every relevant node has performed the precommit phase, but none of them has performed the commit phase.

The node where the two-phase commit protocol is initialized is called the coordinator node for the transactions. Other participating nodes are cohort nodes, respectively. Rubato DB includes measures for dealing with various kinds of failures and/or network partition that might occur. Failures of transactions can lead to aborting itself, and will not lead to blocking the execution of other transactions.

(a) The cohort node times out while waiting for a *pre-commit* message. The cohort can be sure that no *commit/rollback* decision has been taken at any node, since it has not replied any *ready-to-commit* yet. The cohort decides to *rollback*. If a *pre-commit* message subsequently arrives, the cohort will simply respond with a negative response that will prevent coordinator from making a *commit* decision.

(b) The coordinator node times out while waiting for a *ready-to-commit* response from cohorts. The coordinator will decide to *rollback* and send a *rollback* message to all cohorts. Though that all nodes might have sent positive *read-to-commit* response, but one of the replies might not have been delivered during the timeout period. In this case, the coordinator decides to *rollbacks* the transaction, even though all cohorts are operational and reply *ready-to-commit*.

(c) The cohort cannot unilaterally choose to *commit* or *rollback* since the coordinator might have made a different choice, thus violating unanimity. The cohort attempts to communicate with the coordinator to determine the outcome of the transaction. If the cohort cannot communicate with the coordinator, it attempts to communicate with another cohort. If it finds one that has *committed* or *aborted*, it makes the same decision. If it finds one that has not yet reply *read-to-commit*, they both decide to *rollback*. Otherwise, the cohort is blocked until it can determine if the coordinator has made a decision and if so what that decision is.

Similar to the timestamp management approach of Spanner [Corbett *et al.*, 2012] that avoids transactions from being executed with an invalid timestamp, Rubato DB adopts a loading control schema implemented in the socket monitor stage based on the following principles:

At any time, the system should process the requests with least conflict potential. When all the current requests have higher conflict potential, the socket monitor stage would wait awhile for new requests with lower potential to arrive. The conflict potential is evaluated by the number of active clients and priorities among transactions. The socket monitor maintains one list of active clients (with an active transaction) and one list of requesting clients (whose requests are waiting to be processed).

Assume the oldest timestamp in the requesting list is TS_R, and the number of active clients is N_A, and N_O is the number of active clients whose transaction timestamp $T_S \leq TS_R$. The conflict potential at any moment is determined by $\frac{N_O}{N_A}$. If the transaction T_{TS_R} with timestamp TS_R is the oldest among all active transactions, then the ratio is $\frac{1}{N_A}$, and processing T_{TS_R} has the least conflict potential. On the other hand, if T_{TS_R} is the youngest transaction, then the ratio is $\frac{N_A}{N_A} = 100\%$, and processing T_{TS_R} has the highest conflict potential. Rubato DB regulates the socket monitor not process any request when the oldest waiting transaction is not among the older 20%.

5.5 Performance Evaluation

In this section, we provide a detailed experimental evaluation of Rubato DB to analyze the impact of the major design and implementation choices of Rubato DB.

We report various experiments, focusing on the performance evaluation of Rubato DB. The main purposes of experiments are:

1. What is scalability of Rubato DB for the OLTP applications requiring ACID properties?
2. Is Rubato DB capable of handling big data OLTP applications using a collection of commodity servers?

Since Rubato DB is developed using the proposed new formula protocol for concurrency and the staged grid database architecture, the answers to aforementioned questions also provide an assessment to the FPC and the new architecture.

We also present some interesting experiments conducted during the development of Rubato DB that provide much needed insights to FPC and the staged architecture. Particularly, answers are given to the following questions:

3. If numerous conflict operations access data items distributed over multiple nodes in an OLTP application, what is the impact on the performance of FPC?
4. In developing an application using the staged architecture, which is a better choice between using either single-thread or multiple-threads for each stage?

All the experiments reported in this paper use a collection of (up to 16) commodity servers connected with a Gigabit LAN with low network latency. More specifically,

1. Each server has dual quad-core Intel Xeon CPUs (maximum CPU usage is 800%), 32 GB of main memory, SATA disks configured in RAID0.
2. All of the servers are running Linux Ubuntu 12.04 LTS.
3. A Rubato DB server runs on the collection of servers as one database instance.

In order to measure performance of Rubato DB, the standard TPC-C benchmark is used. The **TPC-C benchmark**[4] is a comprehensive database benchmark test

[4]http://www.tpc.org/tpcc/

that continues to be a popular yardstick for comparing OLTP performance on various hardware and software configurations. TPC-C benchmark includes pre-defined database schema to generate warehouse data as the target domain. The TPC-C benchmark was introduced to measure the performance of transaction processing. It is a mixture of read-only and update intensive transactions that simulate the activities found in complex warehouse application environments. The TPC-C benchmark suite consists of nine tables and five transactions that portray a wholesale supplier. The five transactions represent various business needs and workloads:

(a) the NEW ORDER transaction which models the placing of a new order;

(b) the PAYMENT transaction which simulates the payment of an order by a customer;

(c) the ORDER STATUS transaction representing a customer query for checking the status of the customers last order;

(d) the DELIVERY transaction representing deferred batched processing of orders for delivery;

(e) the STOCK LEVEL transaction which queries for the stock level of some recently sold items.

Table 5.4 summarizes the characteristics of the inclusive transactions.

Table 5.4: TPC-C benchmark workloads summary

Transaction	**Workload**	**Operations**	**Frequency**
New Order	Middle-weight Read-Write	Read:50% Write:50%	45%
Payment	Light-weight Read-Write	Read:55% Write:45%	43%
Order Status	Middle-weight Read-only	Read: 100%	4%
Stock Level	Heavy-weight Read-only	Read: 100%	4%
Delivery	Heavy-weigh Read-Write	Read: 40% Write:60%	4%

TPC-C performance is measured in tpmC (i.e., the number of New-Order transactions per minute). The benchmark can be scaled up by increasing the number of warehouses and hence the number of concurrent clients.

Rubato DB's storage manager applies a tree-based schema for grid partitioning. Figure 5.3 shows a representation of the TPC-C schema as a tree schema. The grid partition, as demonstrated in Figure 5.4, is used to deploy data on multiple nodes and thus provides scale-out capability. Tuples in every descendent table are partitioned according to the ancestor that they descend from. As a consequence, all rows inherent from the same root are guaranteed to be co-located and thus data accessed within a transaction will be located in the same data node [Grund *et al.*, 2010; Jones *et al.*, 2010].

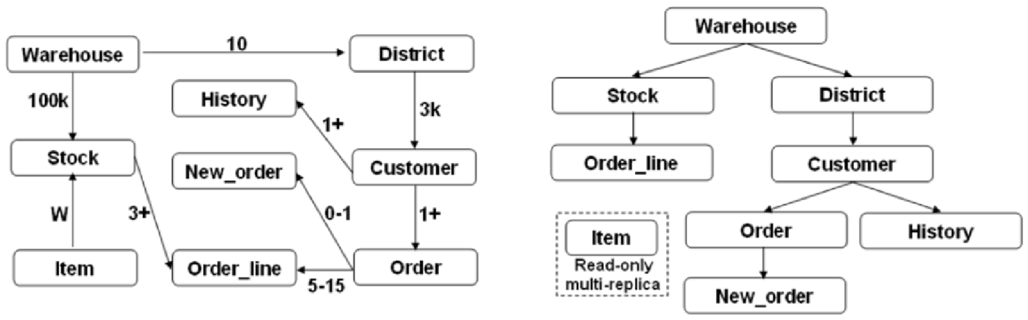


Figure 5.3: TPC-C schema as a tree schema

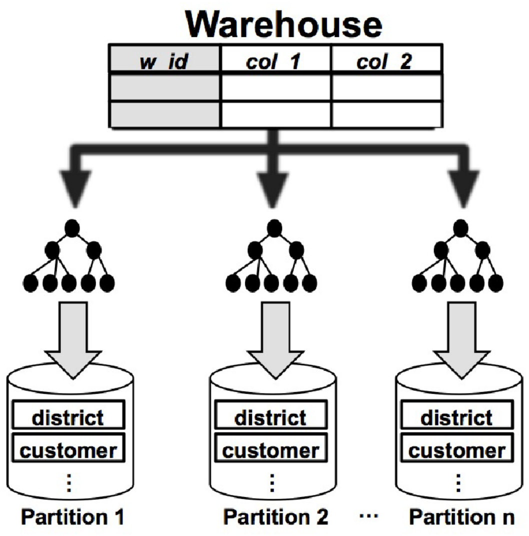


Figure 5.4: Grid partition illustration

5.5.1 Single Node Capacity

We first evaluate the performance of Rubato DB for a single server in isolation, i.e., the transaction manager, sql engine, and storage manager are all setup within a single server. This allows us to analyze the bottleneck for scaling up. We investigate the capacity of a single server system by scaling up the RAM size (i.e., 32 GB, 64 GB, 128 GB). We deploy all staged modules in one server. The test is conducted by increasing the number of clients from 1000 to 20000 (each warehouse has 10 clients as per the TPC-C specification). Figure 5.5 shows the CPUs can achieve high utilization with 128 GB RAM. Considering we only have a collection of commodity servers with 32 GB RAM, so the memory will easily become the bottleneck for fully utilizing the CPU resources. As a reasonable workaround, in the following tests, we strictly comply with TPC-C specification with one exception that is setting 50 clients per warehouse in order to make the best of the computing resources we have. Figure 5.5 shows that our results will still stand if we fully comply with TPC-C specification, i.e., 10 clients per warehouse if we had 16 servers with 128G memory each.

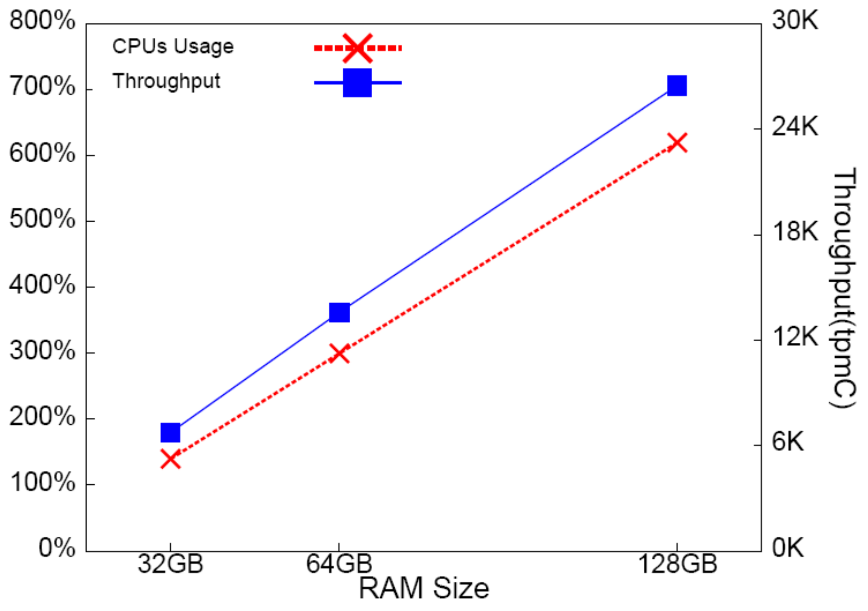


Figure 5.5: Scaling up with Memory Size

5.5.2 Scalability under TPC-C Benchmark

The following set of experiments are conducted to measure scalability of Rubato DB[5] for OLTP applications using the TPC-C Benchmark.

To test the scalability of Rubato DB, we continue the experiments by increasing

[5]By using "SET TRANSACTION ISOLATION LEVEL SERIALIZABLE", Rubato DB provides serializable transaction semantics guarantee.

Table 5.5: TPC-C results

warehouse	size	clients	servers	tpmC
500	72 GB	25000	1	28935
950	135 GB	47500	2	55390
1700	245 GB	85000	4	105572
3200	464 GB	160000	8	184524
6500	943 GB	325000	16	363759

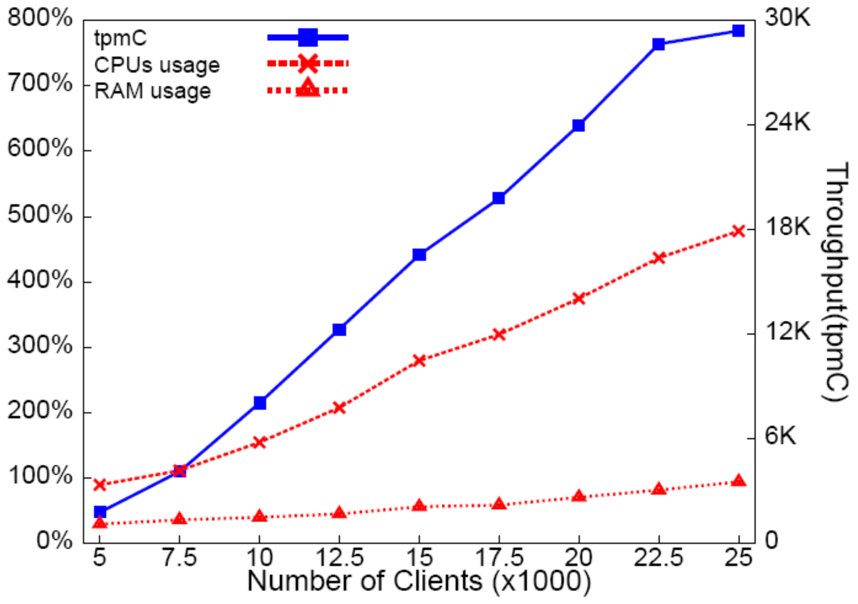


Figure 5.6: TPC-C Performance on 1 node

the number of concurrent clients from 25,000, at the full capacity of one server, to 47,500 using two servers, to 85,000 using four servers, and all the way to 325,000 clients using 16 servers[6]. The latency of all TPC-C tests satisfies the benchmark specification, and as a matter of fact, all transactions are completed within 0.6 second. The performance of scalability is summarized in Table 5.5, and all details are presented in Figures 5.6, 5.7, 5.8, 5.9 and 5.10. There are two types of nodes when the system size is greater than one. Compared with non-main nodes, the main-node also includes an additional socket monitor stage and simulates the client requests of the TPC-C test program[7].

The tests show that the system works smoothly by allocating more computing re-

[6]Rubato DB is currently implemented as a research system in the university lab, and its performance may still be improved dramatically comparing with those commercial ones.

[7]We do not need dedicated client servers for simplicity and minimal hardware cost.

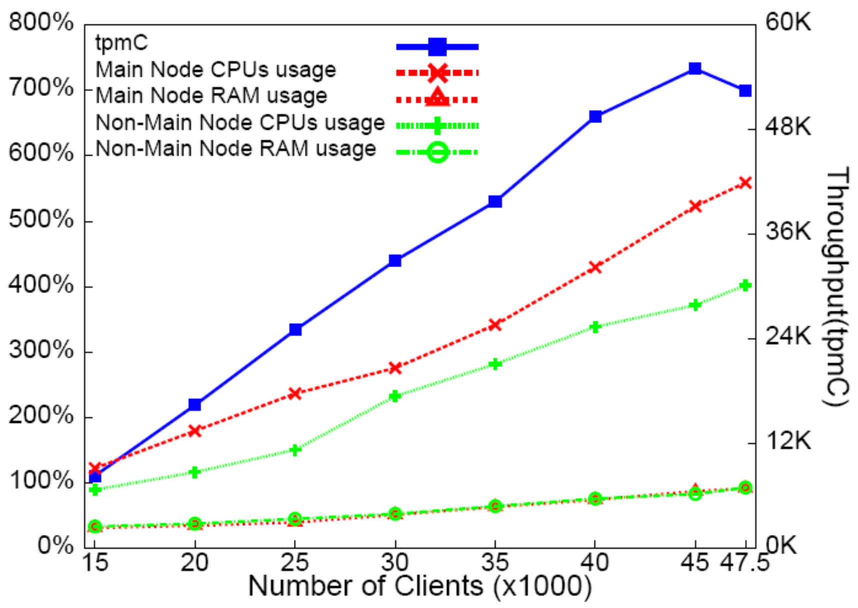


Figure 5.7: TPC-C Performance on 2 nodes

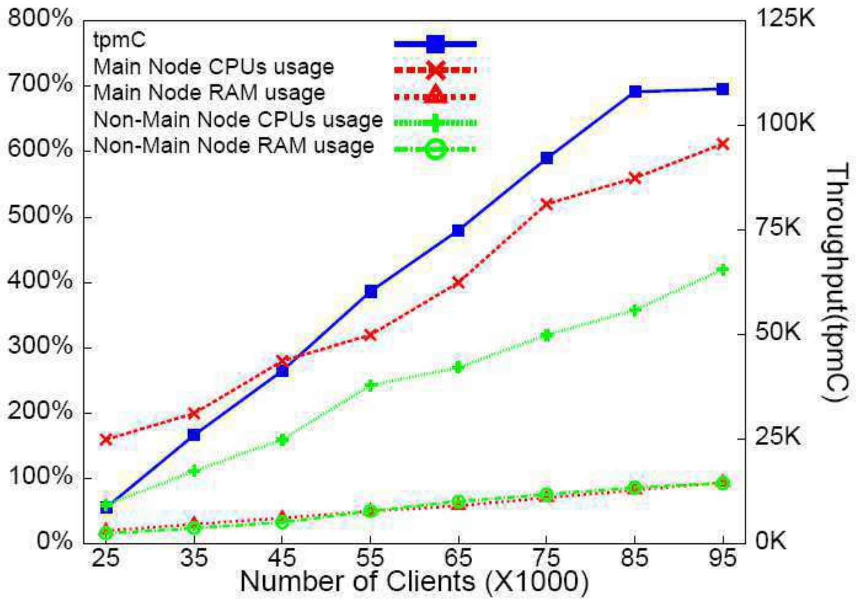


Figure 5.8: TPC-C Performance on 4 nodes

sources gradually to handle growing client base. The CPU and RAM usage percentage increases linearly with the growing client number. However, when the system reaches its full capacity, increasing the number of clients will decrease the tpmC. Then the system needs to scale out by adding more servers. The results clearly demonstrate that Rubato DB is scalable for OLTP applications in that the throughput of the benchmark satisfies a linear growth with the increase of the number of servers used.

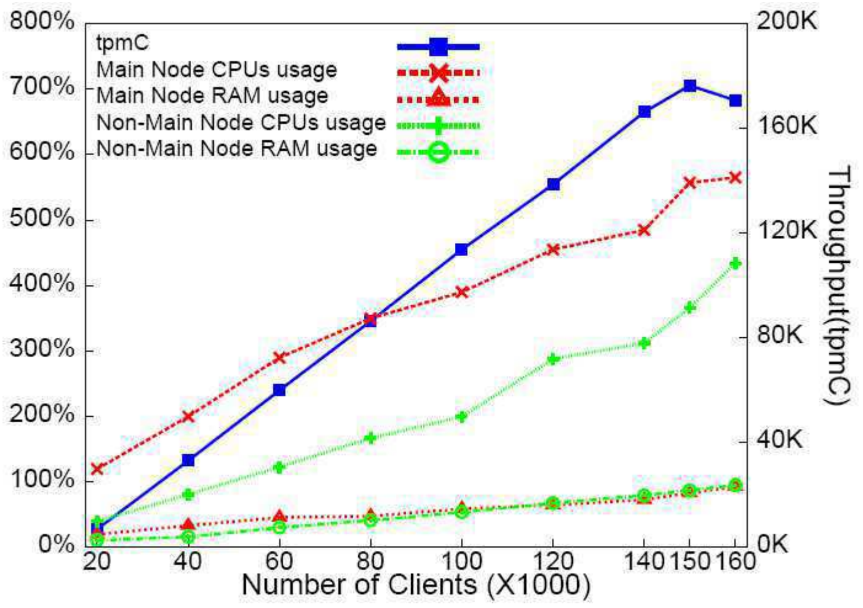


Figure 5.9: TPC-C Performance on 8 nodes

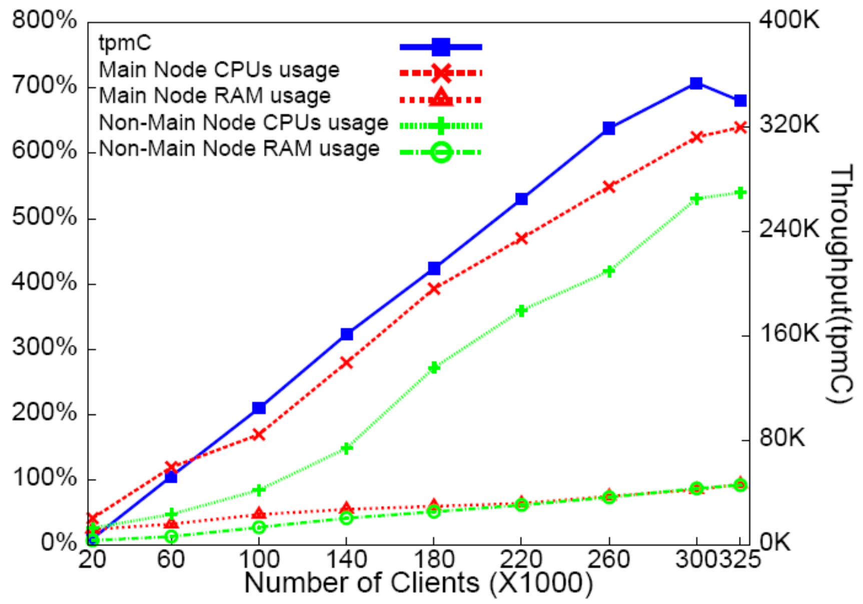


Figure 5.10: TPC-C Performance on 16 nodes

The overall throughput of Rubato DB with different system sizes is plotted in Figure 5.11, which shows the achieved throughput (tpmC) and the rollback ratio on top of the commodity servers cluster. The rollback ratio is stable at a low level of 0.05% as the number of nodes increases. Our experiments also verify that the FPC is scalable in distributed database environment without decreasing the performance. Another interesting observation is that by using the proposed formula protocol for

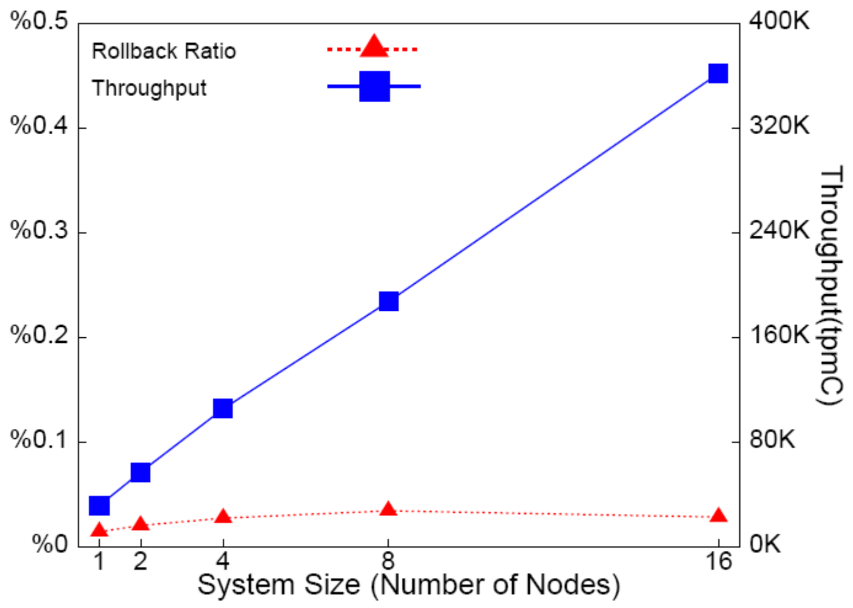


Figure 5.11: Scalability on TPC-C Benchmark

concurrency and the new staged grid database architecture, we can develop a large scale of database applications running on a collection of commodity servers, without using expensive network-attached storage (NAS) systems and/or cluster servers. Thus, in a stable grid environment where all compute nodes works normally, we can predict that Rubato DB is able to achieve superior linear scalability.

5.5.3 Conflict Operations over Different Partitions

This set of experiments are design to investigate the impact on the performance of OLTP applications using the formula protocol for concurrency when a large percentage of conflict operations accessing data items distributed over multiple different grid nodes. We execute a workload derived from the TPC-C benchmark by involving remote guest clients. With remote guest clients, transactions will need to access multiple table partitions across multiple nodes, adding additional network overhead. The performance of the TPC-C benchmark tests with the remote guest accesses varying from 1%, 10%, 20%, and 30% are presented in Figure 5.12.

Rubato DB runs smoothly and correctly[8] with various percentages of remote guest accesses. As expected, the implementation of FPC does not limit the scalability, unless extensive conflicts occur over distributed nodes, as shown by the performance

[8]TPC-C check program is conducted to verify all constraint conditions are passed.

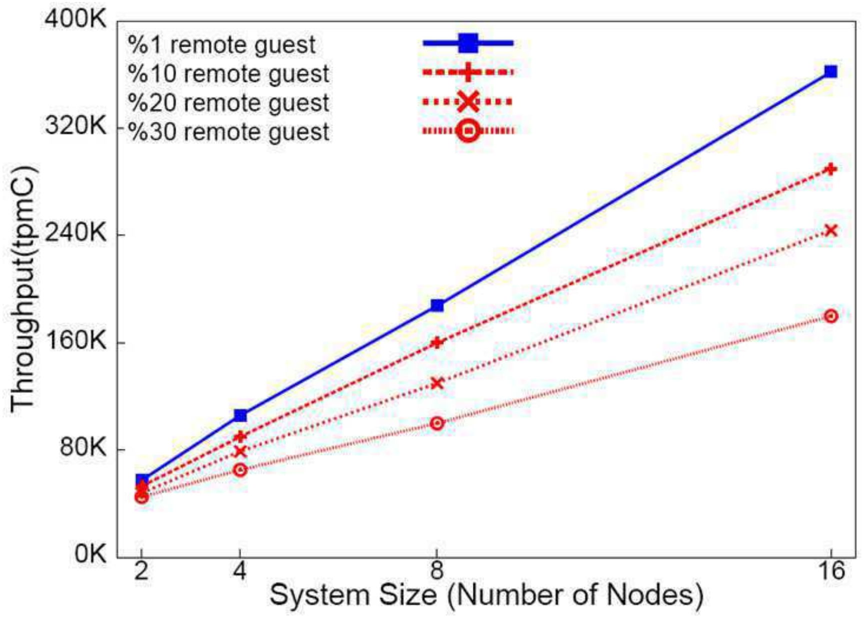


Figure 5.12: Impact of Remote Guest Clients

declining with the increase of remote guest accesses in Figure 5.12: with 10% remote guest accesses, the throughput of the TPC-C test reduces to about 80% on 8 and 16 grid nodes. The performance further reduces to 65% with 20% remote guest accesses. With 30% remote guest accesses, the performance may decrease nearly by half. We believe that the performance repercussions increase relative to the number of nodes mainly because of the communication cost among distributed nodes, which leaves room for improvement of the FPC.

5.5.4 Stages vs. Threads

Modern database systems adopt a thread-based concurrency model for executing query streams. Traditional RDBMS typically use a pool of threads to best utilize the available computing resources. Each incoming query is handled by multiple threads according to the number of available CPUs. Each thread executes until it either blocks on a synchronization condition, an I/O event, or until a predetermined time quantum has elapsed. Then, the CPU switches context and executes a different thread or the same thread takes on a different task. The thread scheduling does not exploit cache contents that may be common across a set of threads. Context-switching typically relies on generated events instead of program structure or the querys current state. It has been reported that the multiple threads may pay a heavy price in context switching, especially in transactional processes involving high code foot-print and exhibiting irregular data access patterns [Harizopoulos and Ailamaki, 2003; Stonebraker *et al.*, 2007].

Different from traditional databases using multi-threads for parallelism, and staged event-driven architecture applications using multi-threads in their stages, Rubato DB is implemented with a single-thread in each and every staged module in its architecture, partially due to the results demonstrated by the experiment below.

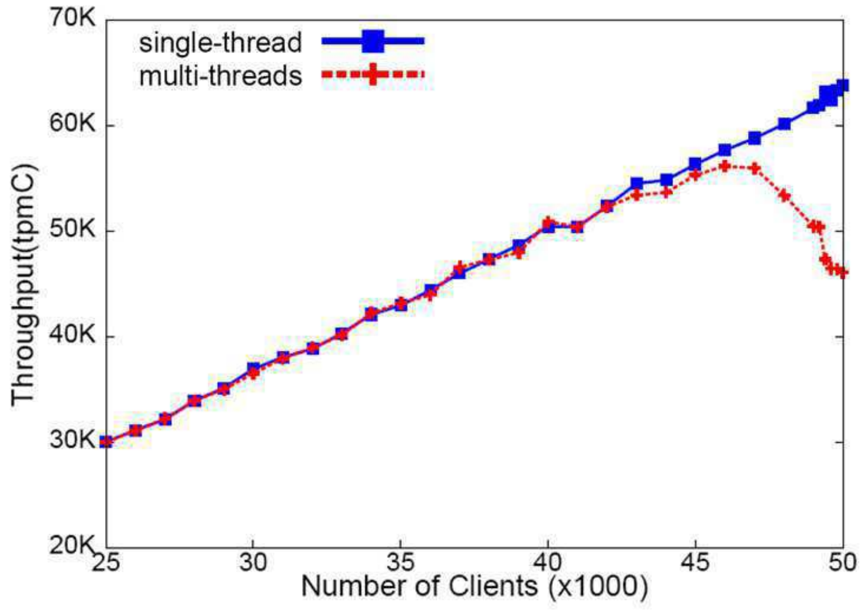


Figure 5.13: Throughput Comparison

We conducted a sequence of experiments for performance comparison of Rubato DB using single-thread stages vs. multi-threads stages running on a collection of 4 commodity servers. The results are presented in Figure 5.13. It clearly demonstrates that the superior scalability of the system using a single thread delivers much better performance than that of the system using multiple threads.

5.6 Chapter Summary

In this chapter, we introduced formula protocol to solve one of the key challenges in designing and implementing NewSQL systems, that is, developing a suitable concurrency control protocol for thousands of concurrent users accessing data distributed over commodity servers. The formula protocol for concurrency is a variation of the multi-version time-stamp Concurrency Control Protocol, which guarantees the serializability of transactions processed by Rubato DB.

To verify the solution of staged grid architecture and formula based protocol for NewSQL system, the performance experiments of Rubato DB are reported in this chapter. The results prove that:

1. Rubato DB is highly scalable and efficient for OLTP applications supporting the ACID properties.
2. Rubato DB is considerably cost effective.
3. It is better to use a single thread for all stages in the staged architecture.
4. The scalability of OLTP applications is not limited by using FPC, unless a large percentage of conflict operations access data items distributed over different grid nodes.

Chapter 6

BASIC: Narrowing the Consistency Gap

"Maintaining a consistent platform also helps improve product support - a significant problem in the software industry."

- Bill Gates.

Based on the understanding of CAP theorem [Brewer, 2002], linearizability is not achievable with high availability in the presence of network partitions, and low latency is not compatible with linearizability either [Gilbert and Lynch, 2002; Bailis *et al.*, 2013*a*]. To minimize latency and remain available during network partition or partial failure, many modern large-scale data management systems eschew transactional functionality, and opt for strong semantic guarantees for atomic non-transactional operations over multiple data partitions (atomic multi-partition operation), such as Google's Bigtable [Chang *et al.*, 2008], Amazon Dynamo [DeCandia *et al.*, 2007], Apache Cassandra [Lakshman and Malik, 2010], Yahoo! PNUTs [Cooper *et al.*, 2008], Linkedin's Espresso [Qiao *et al.*, 2013].

Many of these systems have attempted to provide weaker consistency guarantees - the most notable being BASE (Basic Availability, Soft State, Eventual Consistency) [DeCandia *et al.*, 2007; Lakshman and Malik, 2010; Cooper *et al.*, 2008]. The BASE can be summarized as: the system responses basically all the time (Basically Available), is not necessary to be always consistent (Soft-state), but has to come to a consistent state eventually (Eventual consistency) [Pritchett, 2008]. Despite of its high concurrency and performance benefits, BASE has its own inevitable problems resulting in incorrect behavior for use cases that require consistent visibility for users. Therefore, BASE should not overwhelmingly be the only default choice. We believe the design space for the consistency model is not limited to merely ACID that is too strong and BASE that is too weak, but rather that there is a spectrum between these two extremes, and it is possible to build a set of semantics guarantees combining different consistency models and availability for various use case requirements.

In this chapter, we present BASIC (Basic Availability, Scalability, Instant Consistency), an alternatively higher level of consistency than BASE, for a large underserved class of applications requiring multi-partition, partial-replication, atomically non-transactional operational data access where none of inconsistent (soft) states should be visible to clients. BASIC is desirable because it makes system easier for users to reason about in a scalable manner. BASE and BASIC provide different choices between the model that is faster but requiring extra efforts to deal with inconsistent results and the model that delivers consistent results but is relatively slower with higher latency.

6.1 Motivating Examples

The weak eventual consistency is applicable for some distributed applications requiring “always-on” operations. ATM is such a typical example for eventual consistency, which is designed to have a higher priority for availability than consistency in the partitioning replicated mode. If an ATM is disconnected from the network, it can still provide service to the user. And when the partition eventually heals, the ATM sends a list of requests to the bank and the final balance will still be correct eventually. However, there are a range of large-scale applications that cannot accept weak eventual consistency.

Case 1: Global dependency

For example, in a social networking services (e.g. Facebook’ TAO [Bronson *et al.*, 2013], databases contain information about users’ posts, records, relationships, etc. Maintaining the global dependency between events is an important consideration for building high quality online services [Lloyd *et al.*, 2011; Bailis *et al.*, 2013*b*]. Imagine a hypothetical scenario:

1. Landon posted a status S: “Got injured in a game...”.

2. After the treatment, Landon updated S to a new status S’: “Feel better!”.

3. Landon’s friend Sophia observed S’ and responded J: “Relieved to know that!”.

If merely eventual consistency is respected, a third friend Zoe, can possibly observe J and S rather than S’, causing misunderstanding that Sophia is happy about Landon’s injury. Similarly, foreign-key constraints and bi-directional relationships in the social network (e.g., the *friend_of*, *likes* and *liked_by* association) need to reserve the global dependency (or causality) [Qiao *et al.*, 2013; Bronson *et al.*, 2013; Cooper *et al.*, 2008].

As another example for global dependency, cloud-based health care applications [Wu *et al.*, 2013] prefer value predictable utility and consistency over availability and low latency. Eventually consistent medical record with missing dependency may lead to improper path of patient care.

Case 2: Storage view

We consider an eventually consistent storage system, adopting the ring hashing replication model (e.g., Dynamo [DeCandia *et al.*, 2007], Cassandra [Lakshman and Malik, 2010]) or host based model (Amazon S3 [Brantner *et al.*, 2008], Google F1 [Shute *et al.*, 2013]). Various applications load data into it; meanwhile some others extract data out. When data read from one of the replica, it might succeed, but could be lagged behind for part or all of the data. It is quite difficult to quantify the time differences between pairs of storage operations, since the intervals depend on situational factors (e.g., clock skew, network latency or workload) and configuration parameters (e.g., the number of replicas accessed). So the storage system needs set up certain protocol that requires modifying both the write sets and the read sets. In a more complicated situation, the system has to deal with the fact that reads from non-synchronous tables, which might get cross-reference records that point to data that is invalid or even does not exist [Shute *et al.*, 2013].

Generally, having inconsistent storage view pushes a lot of problems from the storage system to the application. Application developers need appropriate tools to determine what effect different factors have on the value observed by applications, and spend extra work in detecting and compensating for all anomalies that can show up.

Case 3: Partition index

Most large-scale applications might need to perform queries. Local indexes are created so that query requests can be issued against these indexes. Each local index is associated with exactly one partition of the table, which is easy for management but not scalable.

Some systems (e.g. Espresso [Qiao *et al.*, 2013], Cassandra [Lakshman and Malik, 2010], Megastore [Baker *et al.*, 2011], etc.) employ global partition indexes. Global partition indexes are scalable in the degree of partitioning. However, if the eventually consistent fashion is applied, changes may be propagated to the global partitioned indexes after a long delay. Users' applications need to anticipate and handle situations where a query based on these indexes are not correct.

To better understand the problem induced by eventual consistency, we imagine a set of partitioned data items spread over multiple servers. In such distributed infrastructure, network partition may prevent servers from communicating, and in the absence of server failures, communicating messages may also be delayed by factors such as network congestion and routing.

Consider an example of data items[1] with three columns L, S, H distributed over three different partitions P_1, P_2, P_3, respectively; and the following three atomic multi-partition operations on the data item:

1. $a(x)$: $W_a(L = L + x, H = H + x)$;
2. $b(y)$: $W_b(S = S + y, H = H - y)$;
3. $check$: $R_c(L, S, H)$, $assert(L - S = H)$.

Conventionally, we use W and R to denote write and read operations respectively. $a(x)$ increases the value of data items L and H by the value of x; $b(y)$ increases the value of data item S by the value of y, while decreases the value of data item H by the value of y; *check* reads the value of data items L, S, H, and testifies the equation $L - S = H$.

[1]For simplicity, we consider only one data item but all the discussions are valid for a set of data items.

The state of the data item transits from one consistent state to another if and only if it is caused by the completion of an update atomic operation (i.e., $a(x)$, $b(y)$). We assume initial values of L, S, H are all 0.

Example 1. Consider a schedule with no R/W or W/W overlap across all partitions: $S_1 = \{a(20), b(10), check\}$ such that:

1. $a(20)$ completes before $b(10)$ starts;
2. *check* is issued after $b(10)$ is done.

Table 6.1: No R/W or W/W overlap schedule

	schedule			value		
	P_1	P_2	P_3	L	S	H
t_1	$W_{a1}(L+20)$		$W_{a3}(H+20)$	20	0	20
t_2		$W_{b2}(S+10)$	$W_{b3}(H-10)$	20	10	10
t_3	$R_{c1}(L=20)$	$R_{c2}(S=10)$	$R_{c3}(H=10)$			

Table 6.1 describes the schedule on each partition. Note that W_{ai}, W_{bi}, R_{ci} represent the corresponding actions of operations a, b and *check* on the partition $P_i (i = 1, 2, 3)$. The *check* at t_3 returns a consistent state according to BASE, since there exists no overlap among operations on all partitions. However, this is not always the case, as multi-partition operations can overlap due to various reasons (e.g., network delay, network congestion and routing, transmission latency etc.).

Example 2. Consider a schedule with R/W and/or W/W overlap across partitions: $S_2 = \{a(20), b(10), check, check\}$ such that:

1. $a(20)$ is incomplete at t_2 when $b(10)$ starts;
2. the first *check* is issued at t_3 before $b(10)$ is done;
3. the second *check* is issued after $b(10)$ is done.

Table 6.2 describes the schedule on each partition. There exist overlaps between $a(20)$ and $b(10)$; and $b(10)$ and the first *check*.

The schedule in Table 6.2 satisfies BASE, since though the first *check* at t_3 returns an inconsistent state, while at some later time after t_4, the second *check* eventually

Table 6.2: W/W and R/W overlap schedule

	schedule			value		
	P_1	P_2	P_3	L	S	H
t_1	$W_{a1}(L+20)$			20	0	0
t_2		$W_{b2}(S+10)$	$W_{a3}(H+20)$	20	10	20
t_3	$R_{c1}(L=20)$	$R_{c2}(S=10)$	$R_{c3}(H=20)$			
t_4			$W_{b3}(H-10)$	20	10	10
t_5	$R_{c1}(L=20)$	$R_{c2}(S=10)$	$R_{c3}(H=10)$			

achieves a consistent state. For such schedules, clients are required to reason about the correctness of each state which can be consistent or not.

Table 6.3: W/W and R/W overlap schedule for partitioned replicas

	schedule			value		
	P_1	P_2	P_3	R_1	R_2	R_3
t_1	$W_{a1}(R_1 \times (1+20\%))$	$W_{a2}(R_2 \times (1+20\%))$	$W_{b3}(R_3+10)$	120	120	110
t_2	$R_{c1}(R_1=120)$	$R_{c2}(R_2=120)$	$R_{c3}(R_3=110)$			
t_3	$W_{b1}(R_1+10)$	$W_{b2}(R_2+10)$	$W_{a3}(R_3 \times (1+20\%))$	130	130	132
t_4	$R_{c1}(R_1=130)$	$R_{c2}(R_2=130)$	$R_{c3}(R_3=132)$			

Now we consider a case for partial replication. There is a distinction between a fully replicated system, in which all servers are replicas for all data items, and a partial replicated system, in which at least one server acts as a replica for a proper subset of all data items.

Example 3. Consider replicated columns R_1, R_2, R_3 that are distributed across different partitions; and atomic multi-partition operations that are non-commutative[2]:

1. $a(x)$: $W_a(R_1 = R_1 \times (1+\%x), R_2 = R_2 \times (1+\%x), R_3 = R_3 \times (1+\%x))$;
2. $b(y)$: $W_b(R_1 = R_1 + y, R_2 = R_2 + y, R_3 = R_3 + y)$;
3. *check*: $R_c(R_1, R_2, R_3)$, $assert(R_1 = R_2 = R_3)$.

$a(x)$ increases the value of data replicas R_1, R_2, R_3 by the percent of $x\%$; $b(y)$

[2]The order of the execution will affect the final results. Some operations are commutative, such as increment and decrement, of which the execution order does not matter to the result.

increases the value of data replicas R_1, R_2, R_3 by the value of y; *check* reads the value of data replicas R_1, R_2, R_3, and testifies the equation $R_1 = R_2 = R_3$.

Consider a schedule $S_3 = \{a(20), check, b(10), check\}$ with overlaps between $a(20)$ and $b(10)$; and $a(20)$ and the first *check*, as shown in Table 6.3. We assume the initial value for three replicas to be 100.

Both *checks* in Table 6.3 return inconsistent states, even at the time points after t_4. Additional restriction is required for applications to ensure the reconciliation of replicas.

The cases and examples above motivate us to propose BASIC, since BASE has the following inevitable limitations:

- The *eventual consistency* makes only liveness rather than safety guarantee, as it merely ensures the system to be consistent in the future [Lloyd *et al.*, 2011].
- The *soft state* presents challenges for developers, which requires extremely complex and error-prone mechanisms to reason about the correctness of the system state at each single point [DeCandia *et al.*, 2007; Shute *et al.*, 2013; Cooper *et al.*, 2008].
- Additional restriction is required for the *soft state* to converge to *eventual consistency* [Roh *et al.*, 2011; Shapiro *et al.*, 2011; Vogels, 2008].

6.2 BASIC Properties

6.2.1 BASIC Specification

In order to resolve the inconsistency of soft state, we propose BASIC standing for **B**asic **A**vailability, **S**calability and **I**nstant **C**onsistency.

- Basic Availability: the system can response for all continuously operations in a timely manner. The system should never block waiting for an operation that is either delayed or may never arrive in the presence of partitions.
- Scalability: the system is able to scale out by adding more resources for increasing workloads. Partitioned data is arranged to be hosted on distributed servers over a shared-nothing architecture.

- Instant Consistency: all partitioned data seen by each read operation reflects the recent consistent state, i.e., each read operation returns the result that reflects write operations which have been executed completely prior to the read on all partitions.

To define instant consistency formally, we first describe the abstract model over which it operates. Assume the database is running on partitions $P_1, P_2, \ldots, P_n$. Two types of multi-partition operations are considered: (1) READ (i.e., to read a set of data items from multiple partitions) and (2) WRITE (i.e., to write a set of data items into multiple partitions).

An important concept in our model is the notion of dependency between operations. Four rules on each partition define dependency as following:

1. **Direct Read-After-Write** dependency. $w \overset{wr}{\to} r$, if the operation r reads the value written by the operation w.

2. **Direct Write-After-Read** dependency. $r \overset{rw}{\to} w$, if the operation r reads the value that is later overwritten by the operation w.

3. **Direct Write-After-Write** dependency. $w_i \overset{ww}{\to} w_j$, if the operation w_i writes a value that is later overwritten by the operation w_j.

4. **Happen-before** dependency. $op_i \overset{hb}{\to} op_j$, if the operation op_j starts after op_i is finished.

A sub-schedule S_i on the partition P_i is a sequence of operations that access the data item on P_i. A total schedule S is the union of sub-schedules on all related partitions. Given a total schedule S and its all sub-schedule S_i, a direct dependency graph DDG(S) can be constructed by setting a vertex for each operation op in S, and a direct edge $op_m \to op_n$ if any of the previous rules is satisfied.

Following the convention of general isolation level [Adya *et al.*, 2000; Berenson *et al.*, 1995]: **instant consistency** is defined as the absence of the following anomalies in the total schedule S.

- **G0** (Aborted Reads): A total schedule S exhibits anomaly G0 if it contains an aborted write w and a read r, such that r has read a value written by w.

- **G1** (Write Cycle): A total schedule S exhibits anomaly G1 if DDG(S) contains a directed cycle consisting entirely of write-write dependency edges.
- **G2a** (Circular Information Flow): A total schedule S exhibits anomaly G2a if DDG(S) contains a directed cycle consisting entirely of read-write and write-read dependency edges.
- **G2b** (Interference): A total schedule S exhibits anomaly G2b if DDG(S) contains a direct read-write dependency edge from r to w without a happen-before dependency edge from w to r.

A total schedule S is valid under instant consistency iff G0 and G1, G2a, G2b are not present in S and its DDG(S).

- Proscribing **G0** captures the essence of no-dirty-reads anomaly.
- Proscribing **G1** prohibits interleaving of write operations on different partitions.
- Proscribing **G2a** ensures that there is a unidirectional flow of information from op_m to op_n.
- Proscribing **G2b** guarantees that the value in the future should not be returned by any current read.

For instance, in previous Tables 6.2 for schedule {a(20), b(10), check1}:

1. $check1 \overset{rw}{\rightarrow} b(10)$ exists on partition P_2 at t_2-t_3 in $DDG(S_2)$;
2. And $b(10) \overset{wr}{\rightarrow} check1$ exists on partition P_3 at t_3-t_4 in $DDG(S_3)$;
3. Since $DDG(S) = \cup DDG(S_i)(i = 1, 2, 3)$, $DDG(S)$ includes both $check1 \overset{rw}{\rightarrow} b(10)$ and $b(10) \overset{wr}{\rightarrow} check1$, thus **G2a** (Circular Information Flow) is violated.

In the previous Table 6.3 for the schedule {a(20), check1, b(10), check2}:

1. $check1 \overset{rw}{\rightarrow} b(10)$ exists on partition P_3 at t_1-t_2 in $DDG(S_3)$;
2. We do not have $b(10) \overset{hb}{\rightarrow} check1$. The values returned by $check1$ violates G2b (Interference);

3. Until $check2$ began at t_4, we have $b(10) \overset{ww}{\to} a(20)$ on $DDG(S_1)$ and $DDG(S_2)$ for sub-schedules on partition P_1 and P_2, respectively;

4. $a(20) \overset{ww}{\to} b(10)$ exists in $DDG(S_3)$ for sub-schedules on partition P_3. Since $DDG(S) = \cup DDG(S_i)(i = 1, 2, 3)$, it includes both $b(10) \overset{ww}{\to} a(20)$ and $a(20) \overset{ww}{\to} b(10)$, that is against **G1** (Write Cycle).

Instant consistency is weaker than serializability. Instant consistency trades off a lower boundary (operational aligned) of consistency than serializability (transactional aligned) for seeking a higher level of availability and a lower level of latency. Consider two transactions T1={a(20), check1} and T2={b(10), check2}. The serializable schedules include {a(20), check1, b(10), check2} and {b(10), check2, a(20), check1}, both of which also satisfy instant consistency. However, the schedule {a(20), b(10), check1, check2} can only achieve instant consistency only rather than serializability. We present detailed differences in the upcoming section.

6.2.2 BASIC in CAP

The CAP Theorem [Brewer, 2002; Gilbert and Lynch, 2002] asserts that any networked distributed system can have only two of strong consistency, availability and partition tolerance, that is, a highly available system cannot provide strong consistency guarantees in the presence of network partitions. Though strong transactional consistency models are not achievable with availability, many weak consistency models are feasible in the highly distributed environment.

As formally proven [Gilbert and Lynch, 2002], the CAP Theorem pertains to the consistency model of linearizability or the ability to read the most recent write to a data item that is partitioned across servers. However, CAP theorem just provides a relatively narrow scope which is often misconstrued as a broad result regarding the co-existence of consistency and availability in the presence of partitions. By introducing BASIC properties, we are now able to classify a range of models on the three dimensions of consistency, availability and partition tolerance in CAP theorem according to different levels of safety and liveness guarantees.

Serializability, the highest isolation level, guarantees that the concurrent execution of a set of transactions results in a system state that would be obtained if transactions were executed serially, i.e., one after the other. It is typically implemented by pessimistic reads and pessimistic writes, achieving the condition that unless the data is already updated to the latest state, the access to it is blocked. Snapshot isolation is a multi-version concurrency control model based on optimistic reads and optimistic writes. A committed snapshot is taken when the snapshot transaction starts, and remains consistent for the duration of the transaction.

To provide high availability and read scalability, serializability will be combined with replication synchronization, which can achieve one-copy serializability [Bornea *et al.*, 2011]. Within one-copy serializability, the execution of a set of transactions are equivalent to executing the transactions in the serial order within only one single up-to-date copy. Similarly, combining snapshot isolation with replication synchronization, one-copy snapshot isolation can be obtained accordingly [Lin *et al.*, 2005].

Eventual consistency is one of the most commonly used weak consistency models. The eventual consistency guarantees that if no new updates are made to the object, eventually all accesses will return the last updated value. Even though a system with eventual consistency guarantees to converge to a consistenct state, it may not provide a single image system since it makes no promise about the time intervals before the convergence is reached [Roh *et al.*, 2011; Vogels, 2008; Shapiro *et al.*, 2011].

Causal consistency is stronger than eventual consistency by providing additional guarantees on the operations which are causally related. Any execution is the same as if all causally related read and write operations were executed in an order that reflects their causality. Causal consistency guarantees that the execution agrees on the relative ordering of causally related operations [Lloyd *et al.*, 2011; Bailis *et al.*, 2013*b*]. Causal consistency does not prevent conflicting updates [Lloyd *et al.*, 2013].

Table 6.4: Consistency model comparison

		Convergence	Freshness	Causality	High Availability
ACID	One-copy Serializability	Strong	Complete	Yes	No
	One-copy Snapshot Isolation	Strong	Snapshot	Yes	No
BASIC	Instant Consistency	Strong	Snapshot	Yes	Basic
BASE	Causal Consistency	Weak	Arbitrary	Yes	Yes
	Eventual Consistency	Weak	Arbitrary	No	Yes

We compare different consistency models according to the properties of scalability in Table 6.4.

Convergence is usually defined on a sequence of objects. The sequence X_n converges to X, if for every $\epsilon > 0$ there exists an integer m such that for every $n > m$ it happens that $|X_n - X| < \epsilon$. Convergence in the consistency content can be defined as data objects that have received the same updates will reach equivalent state after δ units of time. It is desirable by the systems that δ should be as small as possible.

- Weak convergence. Though data objects that have received the same updates will finally reach equivalent state in the future, it makes no promise about the value of δ.

- Strong convergence. If δ is always smaller than the interval before a read comes after a write, strong convergence can be guaranteed so that a system will not be divergent at the data object for each read operation.

Freshness is also called as bounded staleness [Terry, 2011] based on the quantify staleness of observed read values as the number of missed writes or the amount of time [Cipar *et al.*, 2012; Bailis *et al.*, 2012*b*].

- A consistency model provides arbitrary freshness if the read operation can arbitrarily observe any stale version without restrictions. The arbitrary freshness condition is trivial and undesirable.

- Snapshot freshness guarantees that every read operation observes the most recent snapshot of data committed before.

- Complete freshness guarantees that each read observes the most recent updated data object.

Causality is defined as abstract conditions that ensure processes in a cluster agree on the relative ordering of operations which are causally related. It requires that reads respect the order of causally related writes. With causality, all operations that could have influenced one operation must be visible before the operation takes effect.

Availability

Availability represents a key attribute for large-scale data management systems to guarantee adequate performance ensuring data survival in the event of node failure or network partition.

- Highly available systems ensure "always on" operation and guarantee low latency for single-object, single-operation availability [DeCandia *et al.*, 2007]. We say a system provides high availability if every request that can reach a non-failing server can receive a response from that server, even in the presence of network partitions between servers [Lloyd *et al.*, 2013].

  High availability is considered for single-object, single-operation availability, which is however not sufficient for the applications with operations over multiple objects.

- We say that a system provides basic availability:

  1. If each read operation of a client can access at least one replica for every data object, the read operation will return a response.
  2. If the write operation can achieve at least one replica for every data object it attempts to access, the write operation will commit or reasonably abort. Reasonable aborts ensures that the system can not indefinitely abort operations on its own volition.

Partition Tolerance

Partition tolerance guarantees that consistency is maintained even when network failures prevent some servers from communicating with others. It depends on the policy of update manipulation consisting of update initiation (write-to-one, write-to-many) and update propagation (eager or lazy propagation) [Kemme and Alonso, 2000; Wu and Kemme, 2005].

With write-to-one initiation, all updates are firstly sent to one designate node. This node resolves all conflicts, determines the order to perform updates and broadcasts to other nodes with order preserving [Défago *et al.*, 2004]. The propagation can be eager (i.e., each partition waits to commit until all replicas have received the update, ensuring consistent replicas) or be lazy (i.e., the update on each partition is treated as if it were completed before being applied to all replicas). The lazy propagation is more fault tolerant than the eager propagation, since propagation to each partition is relatively more independent.

The write-to-many initiation is to have W replicas acknowledged the write, thus allowing the write to survive W-1 replica failures [DeCandia *et al.*, 2007]. The system propagates updates to the subset of replicas eagerly and the rest lazily. Quorum-based protocol [Alvisi *et al.*, 2001] is such a typical model of this pattern. Quorum-based protocol is intrinsically partition tolerant, since it depends on majority quorums, operations in any partition that contains a majority quorum will succeed.

Now we extend the current understanding of CAP theorem by characterizing precisely different degrees that can be achieved rather than simply what cannot be done.

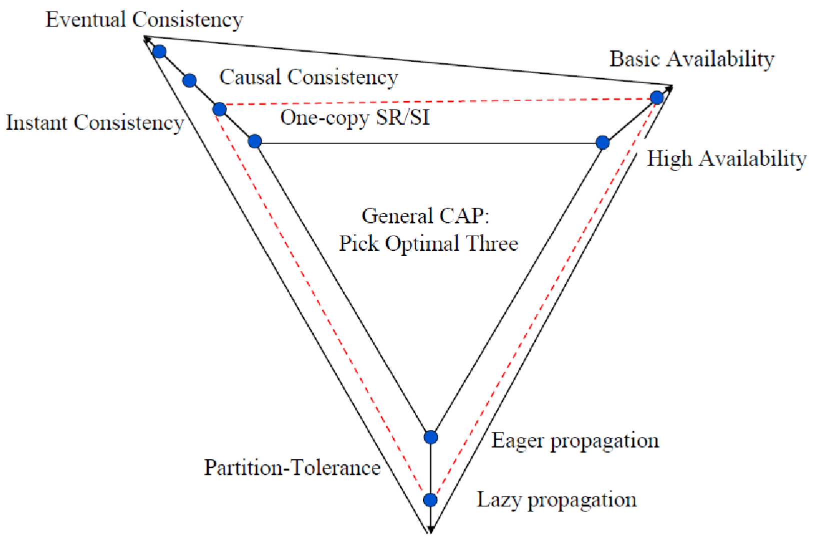


Figure 6.1: General Extension of CAP Theorem

Given the general extension of CAP theorem, as shown in Figure 6.1, we do not just pick two of the three properties of the innermost triangle (analogous to the original CAP theorem), while the optimal three can be chosen based on different application requirements. Thus we can have that a system achievable in BASIC (i.e., the inner dashed line triangle) provides stricter consistency level than a system providing BASE (i.e., the outer solid line triangle).

6.3 Protocol for BASIC

Now we present the protocol for providing instant consistency in multi-partition, partial replicated data management systems. We apply the similar idea in FPC with the optimization.

6.3.1 Protocol for Instant Consistency

At a higher level design, the write operation in the protocol relies on a distributed certification algorithm that is an integration of the two-phase commit protocol combined with the total order multi-casting. The certification is used to validate each write operation and guarantee that once a write updates the data on one partition, all other respective partitions should be updated to the same version. The total order multi-casting facilitates to preserve the order of the commit for write operation among all the replicas of each data item [Bailis *et al.*, 2013*a*; Schiper *et al.*, 2010].

The read operation first acquires the version of the latest available formula from its relevant partitions. Comparing the timestamps fetched by the first round of read, checks are made to verify that if the version of the snapshot visible on each relevant partition is consistent. If any inconsistency is found, the reader needs to issue a second round of read based on a lower version of snapshot, in order to ensure that the according formulas are visible on all relevant partitions. We implement pessimistic and optimistic protocols for read operations based on the tolerance to staleness of snapshot. The pessimistic read must return the most fresh snapshot so that it may need to wait for the completion of write operations. The optimistic read allows potentially stale snapshot to achieve fast response. Now, we present the formal protocol in detail.

Each atomic multi-partition operation *op* is assigned a unique monotonically increasing timestamp, $TS(op)$, when it is initiated. To guarantee partition independence, a list of participating partitions is maintained for each active operation *op*, denoted as $P(op)$. $P(op)$ involves only the partitions that maintain replicas of the data *op* accessed. With each data item x, on each relevant partition P_i, the following pieces of information are stored:

(1) $lrt(x, Pi)$: the largest timestamp of active read operation on data item x on

the partition P_i;

(2) $lwt(x, Pi)$: the largest timestamp of active write operation on data item x on the partition P_i;

(3) $list(x, P_i)$: the list of update formulas.

(4) Initially, $lrt(x, Ni)$, $lwt(x, Pi)$ and $list(x, P_i)$ are set to 0, 0 and $\emptyset$ respectively.

Write operation

To guarantee the atomicity of the write operation, write operation is processed with a distributed certification algorithm integrated in the two-phase commit protocol. The certification is performed over all participating partitions since each partition only knows about the snapshot of its own but not the snapshot at other partitions.

Without loss of generality, consider a write operation W accessing three partitions (P_1, P_2, P_3). Each partition is also replicated by a replica group across sites (multi-partition partial replicated data set), as shown in Figure 6.2. Solid lines indicate inner communication in each replica group, while dashed lines indicates outter communication among leaders and the client. Although the certification adds more messaging for communication, it may impact latency marginally since the requests are processed in parallel. Also to reduce communication overhead, we piggyback information on existing messages, for example, certification result is padded into the prepare_ack message. The distributed certification algorithm is shown in Algorithm 1. The distributed certification algorithm is integrated into the two-phase commit protocol as part of the phase for preparing the operation for acceptance.

Read Operation

Read operation begins by first fetch the latest valid version that is visible for each item from its respective partition. Pessimistic read returns the most recent version having the timestamp smaller than $TS(read(x, P_i))$, no matter the formula is committed or not. Optimist read returns the version with the largest timestamp that is smaller than $TS(read(x, P_i))$, whose status is committed. Comparing the version returned from the first round, the read can check which consistent version of snapshot can be visible from all respective partitions. If necessary, the second round of read will obtain the value according to the version determined in the first round. The distributed read algorithm is shown in Algorithm 2.

Algorithm 1 Distributed Write Operation Algorithm

1. W is sent to any relevant partition (e.g. $P_i, i = 1, 2, 3$) in each replica group.
2. P_i are elected as the leaders for each replica group, respectively.
3. P_i sends the write operation to all cohorts in the same replica group. Certification is done according to rules **W1**, **W2** and **W3** (for pessimistic read) on all cohorts.

   - **W1.** If $lrt(x, P_i) = 0$ or $TS(W(x, P_i)) > lwt(x, Pi)$, $W(x, P_i)$ passes the certification on P_i. A new formula is added into $list(x, P_i)$, and return positive response.
   - **W2.** If $TS(W(x, P_i)) < lrt(x, Ni)$ or $TS(W(x, P_i)) < lwt(x, Ni)$, there must exist at least one *read* (or *write*) operation, which should follow $W(x, P_i)$ according to the timestamp order, has read (written) the value of x before $W(x, P_i)$. Thus $W(x, P_i)$ is too old to write x, thus certification fails. A negative response is returned.
   - **W3.**If $TS(W(x, P_i)) > lrt(x, P_i) > 0$, the write operation is processed as:

     (a) Adding an update formula of $W(x, P_i)$ into $list(x, P_i)$;

     (b) There must exist another operation $R_k(x, P_i)$ such that:

       – $TS(R_k(x, P_i)) < TS(W(x, P_i))$;

       – $R_k(x, P_i)$ has read the value of x before.

     Return a fact $read\_b4(R_k(x, P_i), W(x, P_i))$, indicating $R_k(x, P_i)$ has read the data item x before $W(x, P_i)$. $W(x, P_i)$ must wait for being force-written until $R_k(x)$ is done.

4. P_i aggregates the certification results from all cohorts and acknowledge each other, confirming that its group has done the certification.

   - If all P_i reply positively, it means W will be accepted. Set $lwt(x, P_i) = TS(W(x, P_i))$.
   - If any P_i replies negatively, W will be aborted.
   - Otherwise, W is suspended in the wait queue with a wait_for record according to rule **W3**;

5. P_i collects the response and achieve the final decision to process W. P_i sends the decision (accept, abort or suspend) to all cohorts to take the identical actions for W.

   - In case of acceptance, the formula will be forced written.
   - In case of abort, the formula is removed directly. W does not affect any results.
   - In case of suspend, a reference to the wait-for meta-data is expanded.

6. When W terminates (revoke or force-write), pessimistic read operations that are waiting for W will return.

Algorithm 2 Read Operation Algorithm

1. *read* is sent to any relevant partitions (e.g. P_i, i=1,2,3) in each replica group.

2. On each partition P_i, search $uf(x, op_{u1}, P_i), \ldots, uf(x, op_{um}, P_i)$ in $list(x, P_i)$ such that:
(a) $TS(op_{um}) < TS(read(x, P_i))$ and
(b) $TS(op_{u(m+1)}) > TS(read(x, P_i))$.
(c) Status of $uf(x, op_{um}, P_i)$ is committed (only applicable for optimistic read).

3. P_i acknowledge each other its own $TS(op_{um})$, and the minimal version is chosen.

4. P_i calculates the returned value based on the version determined in Step 3, according to rule **R**.

- **R**. Let $v_0(x)$ be the value of x on the disk of P_i; Let $v_1(x)$ be the value obtained by applying $uf(x, op_{u1}, P_i)$ on $v_0(x)$; $v_{um}(x)$ be the value obtained by applying $uf(x, op_{um}, P_i)$ on $v_{u(m-1)}(x)$. $v_{um}(x)$ is the value to be retrieved by $read(x, P_i)$. $read(x, P_i)$ retrieves the value that is supposed to be obtained by sequentially applying all update formulas on data x issued by operations prior to the read operation.

5. For pessimistic read, $read\_by(op_{uk}, x, Read(x, P_i))$ is recorded for all op_{uk} in $list(x, P_i)$ such that $1 < TS(op_{uk}) < TS(R(x, P_i))$, indicating the value x written by op_{uk} is read by $R(x, P_i)$.

6. Read action:

- If there exists no *read_by*, $read(x, P_i)$ can return immediately.
- Otherwise, $read(x, P_i)$ needs to return after the completion of the op_{uk}. Read is put into the waiting queue with a wait_for record.

7. If $TS(R(x, P_i)) > lrt(x, P_i)$, $TS(R(x, P_i))$ is assigned to $lrt(x, P_i)$, indicating $TS(R(x, P_i))$ is currently the largest timestamp of active read operation on data item x on the partition P_i.

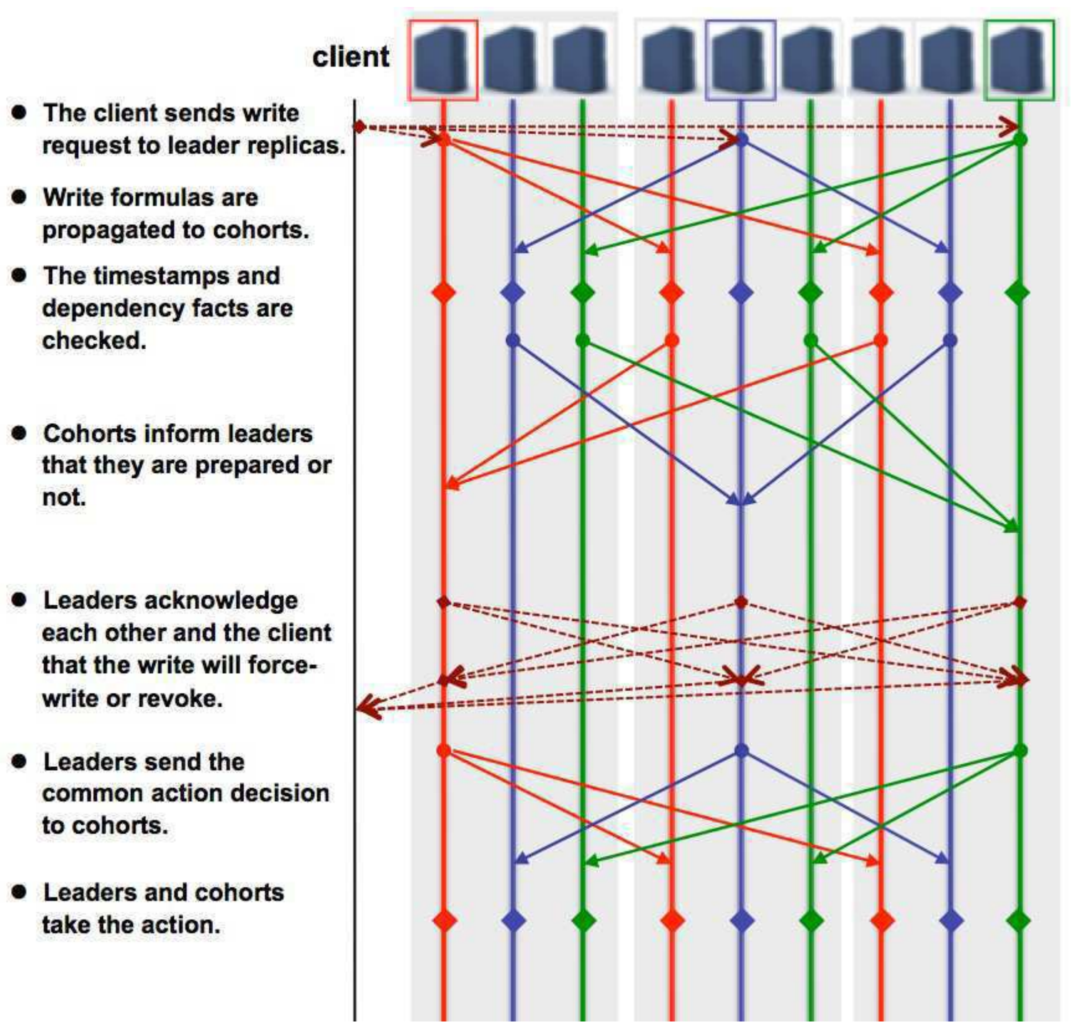

client
The client sends write request to leader replicas.
Write formulas are propagated to cohorts.
The timestamps and dependency facts are checked.
Cohorts inform leaders that they are prepared or not.
Leaders acknowledge each other and the client that the write will force-write or revoke.
Leaders send the common action decision to cohorts.
Leaders and cohorts take the action.

Figure 6.2: Atomic write illustration

Now we use the previous examples to demonstrate how to achieves instant consistency.

Consider the schedule $\{a(20), b(10), check\}$ in Example 6.1 and assume the timestamp for each operation is:

$\{TS(a(20)) = 201, TS(b(10)) = 202, TS(check) = 203\}$.

Operations performed on each partition are illustrated in Table 6.5. The timestamps of the operations are shown as the subscripts of each operation. t_i illustrates different time points during the execution.

(a) At t_1, formulas issued by $a(20)$ on partition P_1 and P_2 are stored in the list of update formulas based on the writing rule **W2**.

(b) At t_2, formulas issued by $b(10)$ on partition P_1, P_2 and P_3 are stored based on the writing rule **W2**, and since there exists no stored facts *read_by* or *read_b*4 associated with $b(10)$, formulas issued by $b(10)$ are then allowed to be force-written to update values for each column. Even though the timestamp of $b(10)$ (=202) is larger than the timestamp of $a(20)$ (=201), $b(10)$ will take effect on the value without waiting for $a(20)$.

(c) At t_3, *check* is suspended due to the existence of stored fact *read_by*.

(d) At t_4, $a(20)$ is revoked since $TS(W_{a3}) = 201 < lrt(H, N_3) = 203$, according to the writing rule **W1**. Upon the removal of formula and *read_by* facts of $a(20)$, *check* suspended at t_3 is resumed by returning a consistent state ($R_{c1} = R_{c2} = R_{c3}$).

Table 6.5: W/W and R/W overlap schedule with dynamic timestamp ordering

schedule				action	value		
	P_1	P_2	P_3		L	S	H
t_1	W_{a1}	W_{a2}		$uf(L, W_{a1}, N_1) \rightarrow list(L, N_1)$			
				$uf(S, W_{a2}, N_2) \rightarrow list(S, N_2)$	100	100	100
				$uf(L, W_{b1}, N_1) \rightarrow list(L, N_1)$			
t_2	W_{b1}	W_{b2}	W_{b3}	$uf(S, W_{b2}, N_2) \rightarrow list(S, N_2)$			
				$uf(H, W_{b3}, N_3) \rightarrow list(H, N_3)$			
				Forcewrite(b) **(dynamic ordering)**	110	110	110
				$R_{c1}(uf(L, W_{a1}, N_1)), read\_by(W_{a1}, L, R_{c1})$			
t_3	R_{c1}	R_{c2}	R_{c3}	$R_{c2}(uf(S, W_{a2}, N_2)), read\_by(W_{a2}, S, R_{c2})$			
				$R_{c3}(H = 110), lrt(H, N_3) = 203$			
				Revoke(W_{a3}), Revoke(a)			
t_4			W_{a3}	$uf(L, W_{a1}, N_1), read\_by(W_{a1}, L, Rc_1)$			
				$uf(L, W_{a2}, N_2), read\_by(W_{a2}, L, Rc_2)$			
				$R_{c1}(L = 110), R_{c2}(S = 110), R_{c3}(H = 110)$	110	110	110

Instant Consistency Guarantee

We now consider how the protocol preserves instant consistency according to the definition (i.e. proscribing G0, G1, G2a, G2b).

- **G0** (Aborted Reads): The two-phase atomic commit protocol of write operation can guarantee that once op_{uk} is applied on one partition, all other respective partitions should be updated to the same version. Otherwise if op_{uk} aborted, the formula $uf(x, op_{uk}, P_i)$ will be eliminated so that $read(x, P_i)$ can not see the updates according to rule **R**.

- **G1** (Write Cycle): If there exist w_i and w_k such that $w_i \rightarrow w_k$, then we have $lwt(x) = TS(w_k) < TS(w_i)$ according to rule **W2** and Step 4 in write algorithm. Similarly, if there also exists $w_k \rightarrow w_i$, we will have $lwt(x) = TS(w_i) < TS(w_k)$, inducing the contradiction.

- **G2a** (Circular Information Flow): If $r_i \rightarrow w_j$ exists, it means that a stored fact $read\_b4(r_i, w_j)$ will be generated according to rule **W3**. The $read\_b4$ fact guarantees that w_j with a larger timestamp can not be issued before r_i with a smaller timestamp. Therefore, it is impossible that r_j read a version produced by w_i, so that $w_i \rightarrow r_j$ can not happen according to step 2 in the read algorithm.

- **G2b** (Interference): If there exists $w_i \rightarrow r_j$ means that r_j has read a version produced by w_i. If w_i with smaller timestamp arrives later than r_j, w_i will abort and retry, since w_i arrives too late to force-write according to the writing rule **W1**.

  Otherwise, a *read_by* fact, according to the step 5 in the read algorithm, is generated and regulates $TS(w_i) < TS(r_j)$.

6.3.2 Basic Availability Guarantee

The protocol operates in a distributed environment, which needs to deal with partial failure and network partitions. Partition independence ensures that failed clients do not cause other clients to fail. This provides fault tolerance and availability as long as clients can access relevant working partitions.

The write protocol use a two-phase atomic protocol, which will always complete the operation except when every relevant partition has performed the first certification phase, but none of them has performed the second phase. Particularly,

- (i) If a partition P_r accessed by an operation *op*, has timed out while waiting during the certification phase, P_r can abort the operation and safely discard its formula, since P_r can be certain to never force-write *op* in the future.

- (ii) If a partition P_r has performed the certification phase but times out while waiting in the second phase, P_r can check the status of operation *op* on any other partitions to determine the outcome of the operation. If another partition has executed force-write for *op*, then P_r can force-write *op*.
- (iii) The leader unilaterally aborts the operation if it times out while waiting for other cohorts for the certification phase.
- (iv) Every event happened is logged on each partition, including all operation submission and phase status changes. If the leader becomes unavailable in each replica group, another cohort takes over. The replica group is available as long as there is one available replica to respond.
- (v) If all of the cohorts in the group go down, all running operations are lost. Operations that were running are assumed to have exited.

Failures of write operations can lead to aborting itself, and will not lead to blocking the execution of read operations.

Partition independence ensures that one operation only contacts partitions that its operation accesses, and operations accessing totally different group of partitions do not affect each other. The partition independence can reduce the work burden of partitions that are not directly involved in an operation's execution. And it is important in the presence of partial failure that prevents one client's operation from causing another to block.

6.4 Performance Evaluation

In this section, we are going to conduct experiments on the performance evaluation of BASIC. The main purpose of experiments includes:

1. What is the inconsistent soft states ratio in the systems with BASE ?
2. What are the performance comparisons between systems with various consistency models?
3. What is the scalability for systems with BASIC?
4. How much will the latency of response increase for systems with BASIC?

6.4.1 Experimental Setup and Benchmark

In our experiments, we compare different system configurations including: Rubato DB with BASIC, Rubato DB with BASE, HBase[3] with BASE, and Cassandra[4] with BASE. We run all our experiments on a collection of, up to 48 partitions on 12 Cybera[5] instances. In order to measure the performance of those systems, two standard commonly used benchmarks YCSB and TPC-C are applied.

The Yahoo Cloud Serving Benchmark (YCSB version 0.1.4) [Cooper *et al.*, 2010] is a data serving benchmark widely used to measure throughput and latency with varying operation distribution for big data systems. According to the YCSB specification, we define a multi-column structure for each data item, which consists of 48 data columns. We use column partitioning and distribute columns across multiple servers evenly. We test the read-intensive workload (including 90% read operations and 10% write operations) and the write-intensive workload (including 50% read operations and 50% write operations). In the experiments, a continuous mixed workload is submitted into the system, and the benchmark then measures the performance in terms of throughput (i.e., the number of operations per second) and latency of operations in milliseconds.

6.4.2 Inconsistency Ratio

We first calculate the inconsistent soft states ratio of BASE on the number of partitions as 2, 4, 8, 16 and 24. Keys of data are hashed uniformly across all partitions. The keys are drawn from "hot spot" distribution, with 80% of the operations going to a subset of hot keys comprising 20% of the key space. We evaluate the inconsistency of value returned by read operations applied to Rubato DB with BASE and Cassandra with BASE.

The inconsistent state of read operation in Rubato DB is detected by checking the update formula on each participating nodes. If the formulas employed by the read operation do not coincide, then the state returned is inconsistent.

Cassandra version 2.0.9 is installed and configured with default parameters on

[3]http://hbase.apache.org/
[4]http://cassandra.apache.org/
[5]http://www.cybera.ca/projects/cloud-resources/rapid-access-cloud/

all servers. One additional server is used to simulate multi-threaded YCSB clients, 32 client threads per YCSB process, and 80% read and 20% write operation mix. Cassandra is configured with a replication factor of 3. By default, the Cassandra consistency level is "ONE" for both reads and writes, which requires that a write be applied to the commit log and memory table of at least one replica before returning acknowledgment to the client. The read return the value obtained from the first replica that responds. The value returned by the read is compared with the latest write value. If they are not equal, we regard the read to be inconsistent.

Table 6.6: Inconsistent Soft State Ratio

Partitions	2	4	8	16	24
RubatoDB	%0.08	%0.15	%0.19	0.28%	0.42%
Cassandra	%0.04	%0.12	%0.29	0.45%	0.63%

As the aggregate number of YCSB clients is proportional to system size, we observe that the consistency violations become more severe as the number of servers increases. Table 6.6 shows that the inconsistent soft state ratio of Rubato DB on the 2-node system is lower than 0.1%; however, the ratio increases gradually with the growth of the system size, which is as high as 0.42% on the system with 24 nodes. The inconsistency ratio in Cassandra on the 2-node system is about 0.04%, and all way increases to more than 0.6% on the system with 24 nodes. Such increasing tendency of inconsistency indicates that more efforts are required to deal with the soft state as the system scales out. As the system size becomes large, it is clear that uncertainties in communication may prevent all partitions from drawing accurate decision about the instantaneous global state of the system. Another source of inconsistency may also arise if all partitions fail to execute identical reactions or evaluate the same value.

6.4.3 Scalability on YCSB

We now compare the performance in terms of throughput and latency as per the YCSB specification. In this group of experiments, we set up three replicas for each data item, that is using HBase (replication factor=3) and Cassandra (replication factor=3).

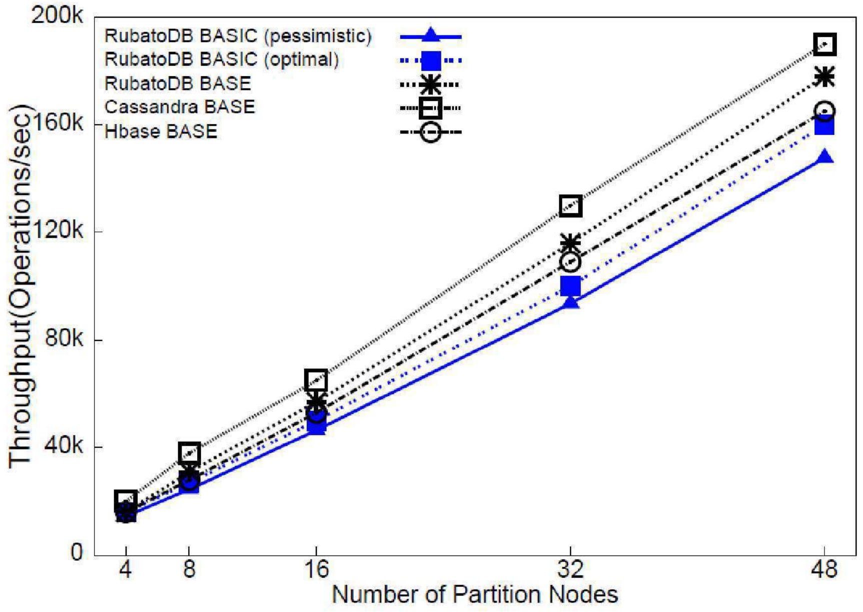
Our results show that the throughput and latency of Rubato DB with BASE is comparable with Cassandra and HBase, but Rubato DB with BASIC has potentially lower the performance and higher the latency. The throughput comparison for read-intensive (write-intensive) workload is plotted in Figure 6.3(a) (Figure 6.4(a)) and the corresponding latency is presented in Figure 6.3(b) (Figure 6.4(b)).

In the read-intensive workload, the number of conflict operations and update formulas on each partition is quite limited. Most read operations are able to response efficiently with little effort to obtain the converged consistent version of snapshot. Thus, the second round of most read operations can be skipped. And also since the concurrency of write operations is relatively low, the write certification process will return positive result with higher possibility, since rule **W1** is applied for most cases. With 48 partition, the throughput of optimal and pessimistic read implementation of BASIC is quite similar, which decreases about 10% than BASE. The latency increases by 1.2 times.

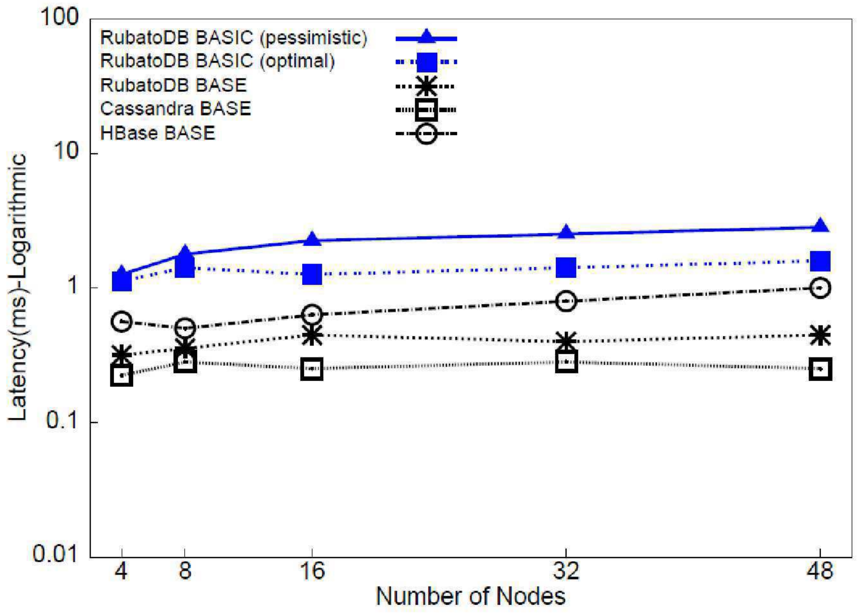
In the write-intensive workload, the potential of operation confliction increases, causing a higher cost for ensuring BASIC. As illustrated in Figures 6.4(a) and (b), the throughput reduces 25% with the latency increasing nearly one order of magnitude. The main source of the latency comes from the presence of abundant dependencies between operations. As more cases trigger rules **W2** or **W3**, the numerous stored facts (i.e. *read_by*, *read_b4*) can cause one operation to wait for response until another operation is done. At the same time, as the update formula list becomes longer, it takes more time to calculate the final value for each snapshot according to the read rule **R**.

Though there is decline for the performance, we can still find that Rubato DB with BASIC preserves near-linear scalability with increasing throughput and flat latency, same as systems with BASE.

In addition, as shown in Table 6.7, though the rollback ratio rises as the number of partitions increase, it is still quite stable at a lower level than 0.1% for the read-intensive workload and 1% for the write-intensive workload as the number of partitions increases. Thus, it ensures a reasonable rollback ratio range for the basic availability guarantee.

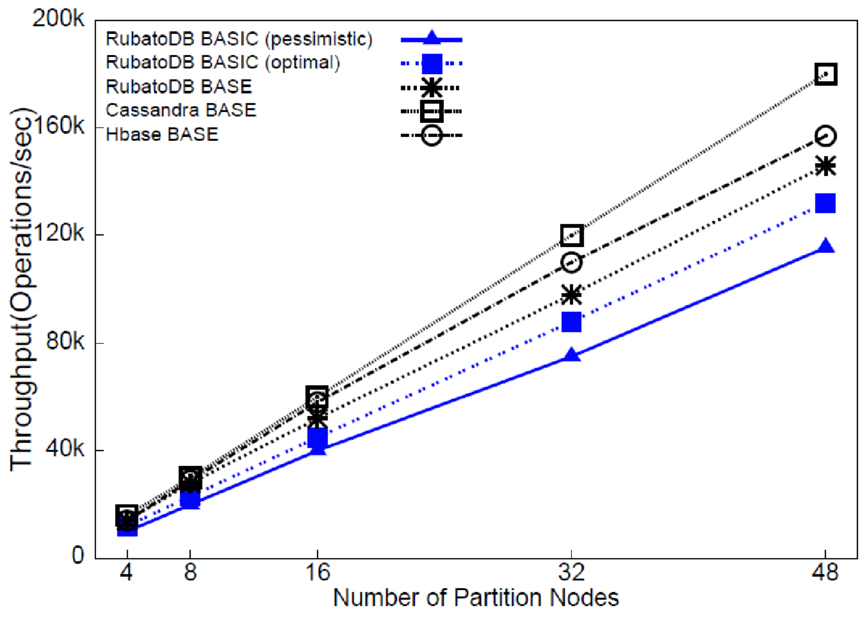


(a) Read Intensive Workload Throughput

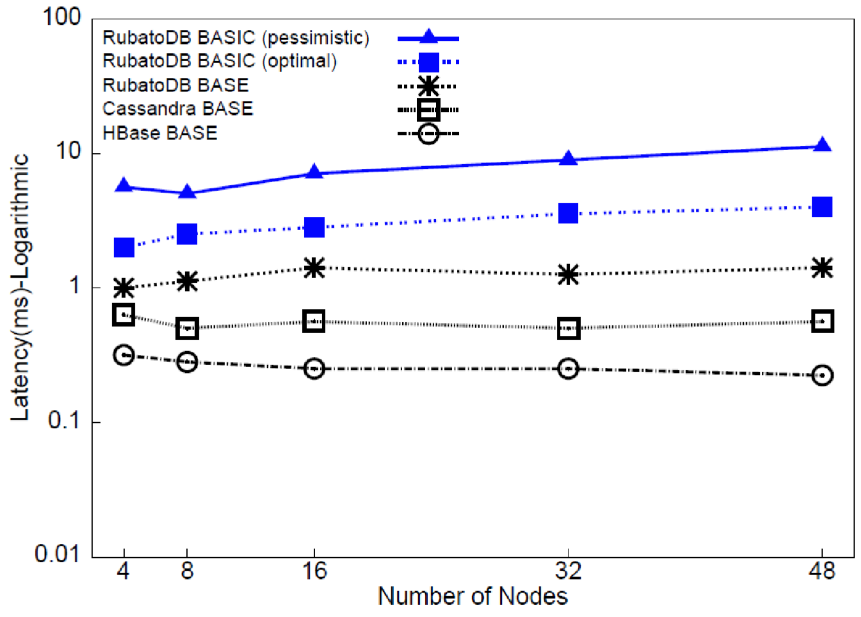


(b) Read Intensive Workload Latency

Figure 6.3: Comparison between BASIC and BASE for read intensive workload



(a) Write Intensive Workload Throughput



(b) Write Intensive Workload Latency

Figure 6.4: Comparison between BASIC and BASE for write intensive workload

Table 6.7: Rollback Ratio

Partitions	4	8	16	32	48
Read-intensive	%0.01	%0.017	%0.03	0.054%	0.093%
Write-intensive	%0.05	%0.12	%0.37	0.62%	1.1%

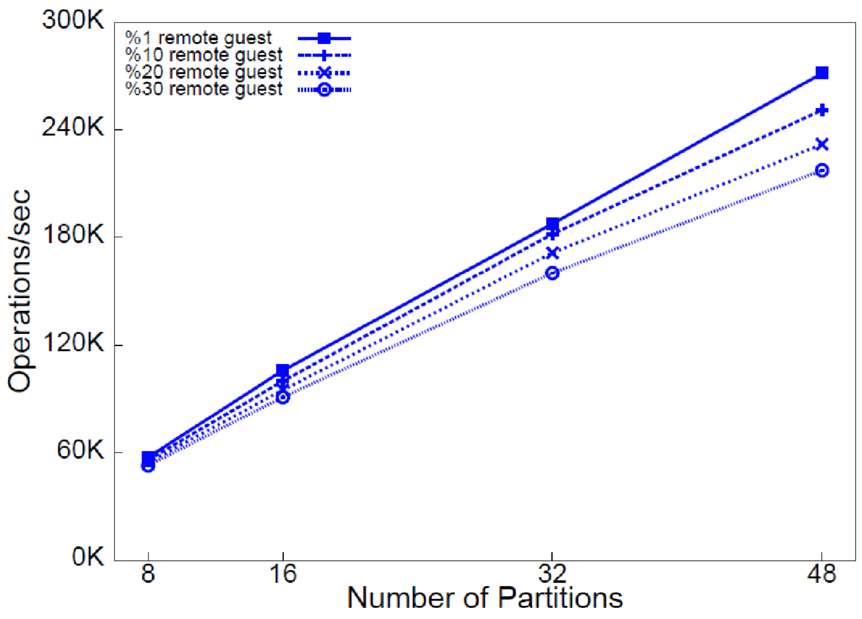


Figure 6.5: Comparison of Scalability on TPC-C

6.4.4 Scalability on TPC-C

This set of experiments are design to investigate the scalability on the performance of OLTP applications complying with BASIC properties[6]. We also apply a tree-based schema to partition TPC-C data across multiple partitions, since this schema structure is efficient for splitting and merging of partitions. With remote guest clients, operations will need to access multiple data partitions across multiple nodes, requiring multi-partition operations over the cluster.

The performance of Rubato DB with BASIC evaluates the TPC-C benchmark tests with the remote guest accesses varying from 1%, 10%, 20%, and 30% are presented in Figure 6.5.

As expected, the system with BASIC degrades the scalability, especially when extensive remote guests occurs. However, the decline is minimal. As shown by the

[6]Since BASIC is weaker than ACID, serializability is not guaranteed. The TPC-C test may fail to pass conditions in tpccheck.

performance declining with the increase of remote guest accesses in Figure 6.5: with 10% remote guest accesses, the throughput of the TPC-C test reduces to about 8% on 48 partitioning nodes. The performance further reduces to 15% with 20% remote guest accesses. With 30% remote guest accesses, the performance may decrease around 20%. The performance degradation is mainly because transactions in the TPC-C benchmark have a larger data access scope, which will induce more dependencies among operations. And as the remote guest ratio increases, the data skew issue [Xu *et al.*, 2008; Kossmann *et al.*, 2010] will become more severe, causing some partitions to be hot spots.

On the other hand, the degradation is quite minimal, which can partially be attributed to the fact that systems with BASIC will commit or rollback operations as soon as they are issued on all participating partitions rather than being postponed to the end of each transaction. Such immediate action for each operation can minimize the overhead induced by remote guest accesses.

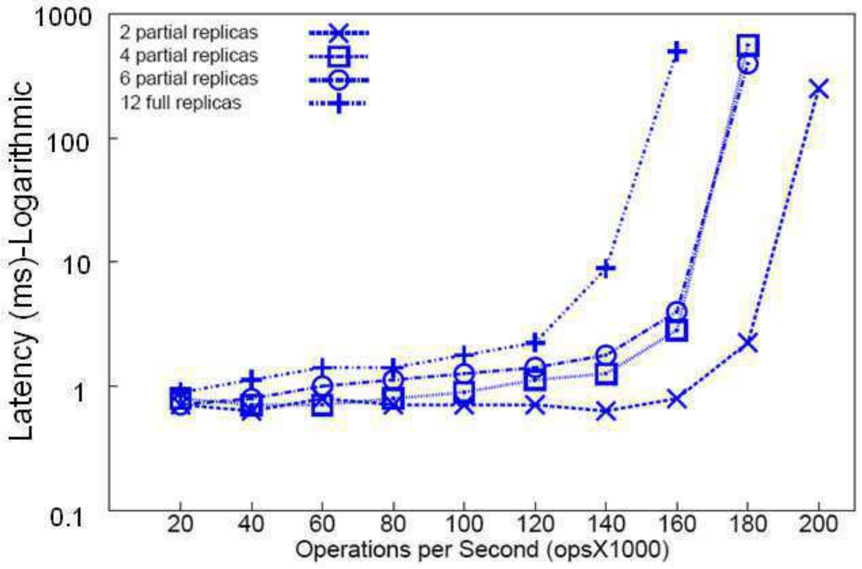
6.4.5 Increasing Replication

The following experiments explore the response latency under increasing workloads of partial and full replication. A 12-node Rubato DB with BASIC is used as the test bed. We increase the replication factor from 2, 4, 6 for partial replicas, to 12 for full replicas. Figure 6.6 shows the latency of response time for 12-node Rubato DB with BASIC under various replication factors.

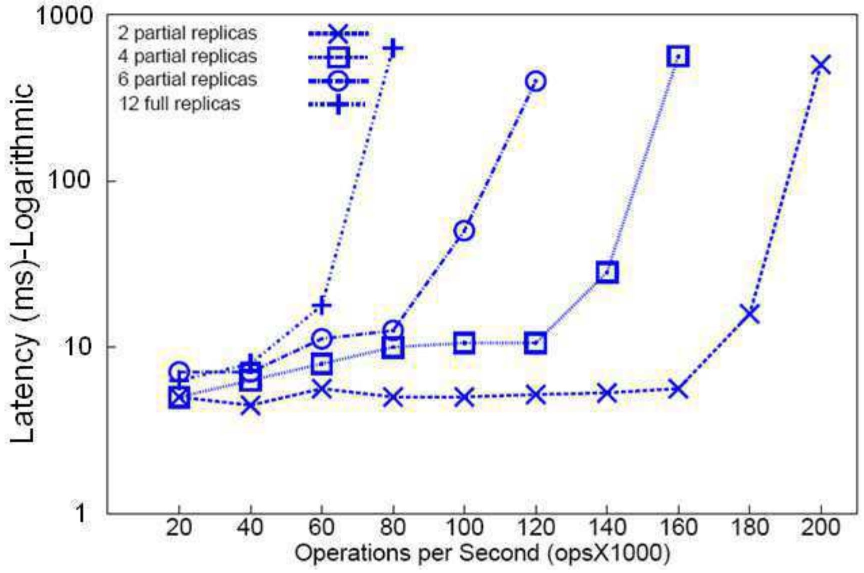
The partial replication results (2, 4, and 6 replicas per data item) show that the smaller the number of replicas is, the higher throughput (operations per second) can be achievable with flat low latency. It is reasonable that less cost is required for synchronization with a smaller the replication factor.

Particularly, since the read-intensive workload induces less overhead for write certification, the response time of full replication stabilizes for 70% of maximum workloads, and partial replication can achieve 85% of the maximum workloads before the latency rockets (as shown in Figure 6.6(a)). The latency of replication does not decline the performance of the system quite much, since that results of any read distributed across replications can obtain consistent snapshot easily.

In the write-intensive workload, a "safe" write operation requiring multiple repli-



(a) Read Intensive Workload



(b) Write Intensive Workload

Figure 6.6: Comparison of Latency for Replication

cations acknowledge the write before it returns will incur higher latency, and/or may time out in presence of network partition. Requiring confirmation of each write operation has replicated to all partitions will effectively guarantee that those replicated partitions have caught up with the timestamp of this write. The partial replication can scale significantly better than full replication (as shown in Figure 6.6(b)), but still the more replicas are used, the more dependency stored facts are involved causing higher latency. This exhibits that the latency of operations increases with high contention. The clients wait longer for the majority response for either abort or acceptance. When the number of operations per second is low ($< 40,000$), the latency does not vary as much as the replications increases. This is because when the contention is not that high, the latency can be maintained low with more rollback operations; an abort operation exhibits lower latency due to its early termination compared to a committed operation.

In summary, for the read-intensive workload or the write-intensive workload without requiring extremely low response latency, the cost induced by BASIC is acceptable comparing with the extra efforts needed to manipulate inconsistent soft states for BASE. That is, BASIC pays a reasonable price for a higher consistency than BASE.

6.5 Chapter Summary

We explicitly identify important disadvantages of BASE with tangible examples and formally define BASIC property, which provides an alternatively higher level of consistency than BASE, while lower than ACID. BASIC is desired by a large class of applications requiring multi-partition, partial-replication, atomically non-transactional operational data access where none of inconsistent states should be visible to clients. We generalizes the understanding of CAP theorem based on BASIC. Given the general extension of CAP theorem, we do not just pick two of the three properties, while the optimal three can be chosen based on different application requirements.

Chapter 7

Summary and Conclusions

"We can only see a short distance ahead, but we can see plenty there that needs to be done."

- Alan Turing

7.1 Contributions

Over the past few years, big data has emerged as a multi-billion dollar industry and as a successful paradigm for large scale data management systems deployment. Since DBMSs store and serve an application's critical data in the past, making them adapt the big data application become a critical mission. NewSQL database management systems deployed in a scalable infrastructure aim to provide such solutions. The overarching goal of this dissertation was to make NewSQL systems scale-out while efficiently supporting transactional semantics and being efficient without introducing high performance overhead. This dissertation makes fundamental contributions in the thrust areas of principles, paradigms and protocols to architect and implement practical NewSQL database systems. These contributions are critical to the design and implementation of NewSQL database systems and significantly advances the state-of-the-art in the field.

In particular,

(1) We categorize and analyze the critical aspects of NewSQL database systems, comparing with NoSQL systems. We propose taxonomies to classify techniques in which every high scalable system is able to find its position. A thorough understanding

of current systems and a precise classification are essential for analyzing the scalability limitations.

(2) We specify a staged grid architecture and abstractions to support scalable and efficient transaction processing using clusters of commodity servers. The key insight is to disintegrate and reassemble system components into encapsulated staged modules. Effective behavior rules for communication are then defined to orchestrate independent staged modules deployed on networked computing nodes into one integrated system.

(3) We propose a new formula-based protocol for distributed concurrency control to support for thousands of concurrent users accessing data distributed over commodity servers. The formula protocol for concurrency is a variation of the multi-version time-stamp concurrency control protocol, which guarantees the serializability. We reduce the overhead of conventional implementation by technologies including logical formula caching and dynamic timestamp ordering.

(4) We identify a new consistency model - BASIC (Basic Availability, Scalability, Instant Consistency) that matches the requirements where extra efforts are not needed to manipulate inconsistent soft states of weak consistency models. BASIC extends the current understanding of CAP theorem by characterizing precisely different degree of dimensions that can be achieved rather than simply what cannot be done.

We introduce all these features based on a novel implementation of Rubato DB, a highly scalable NewSQL database system. We have conducted extensive experiments that clearly show that Rubato DB is highly scalable with efficient performance under both TPC-C and YCSB benchmarks. These results verify that the formula protocol and the staged grid architecture provide a satisfactory solution to one of the important challenges in the NewSQL database systems: to develop a highly scalable database management system that supports various consistency levels from ACID to BASE.

7.2 Directions for Future Work

The continued growth of data sizes, advent of novel applications, and evolution of the infrastructure ensures that the area of data management facing the big data trends has many interesting research challenges. Based on the techniques presented in this dissertation, some direct extensions will be worthwhile for further research.

(1) Access driven hybrid data partitioning techniques depend on the applications access patterns to partition data to avoid data skew and hot spot. As the application's access patterns modify, it might also need re-partitioning. Traditional OLAP approaches rely on long unavailability waiting time to repartition the data and are therefore not amenable to support regular re-partitioning. The challenge is to re-partition the data real-time while minimizing service interruption.

(2) Administering large scale database systems is expensive. Automatic administration of a large cluster of servers minimizes the need for human intervention for resource orchestration. The responsibilities of such a self-managing controller include monitoring the behavior of the system, dynamically scaling out based on current bottleneck of the performance. The goal is to ensure that the performance guarantees are met while ensuring effective resource utilization.

(3) Extending the designs of NewSQL database systems to the dynamic cloud infrastructure with larger scale operations, higher network latency, and lower network bandwidth is also a worthwhile direction of future work.

Bibliography

Abadi, Daniel J, Samuel R Madden and Nabil Hachem (2008). Column-stores vs. row-stores: How different are they really. In: *Proceedings of 2008 ACM SIGMOD International Conference on Management of Data*. ACM. pp. 967–980.

Abadi, Daniel, Samuel Madden and Miguel Ferreira (2006). Integrating compression and execution in column-oriented database systems. In: *Proceedings of 2006 ACM SIGMOD International Conference on Management of Data*. ACM. pp. 671–682.

Abouzeid, Azza, Kamil Bajda-Pawlikowski, Daniel Abadi, Avi Silberschatz and Alexander Rasin (2009). HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for analytical workloads. *Proceedings of the VLDB Endowment* **2**(1), 922–933.

Adya, Atul, Barbara Liskov and Patrick O'Neil (2000). Generalized isolation level definitions. In: *Proceedings of the 16th International Conference on Data Engineering*. pp. 67–78.

Agrawal, Divyakant, Sudipto Das and Amr El Abbadi (2011). Big data and cloud computing: current state and future opportunities. In: *Proceedings of 2011 ACM International Conference on Extending Database Technology*. ACM. pp. 530–533.

Agrawal, Sanjay, Vivek Narasayya and Beverly Yang (2004). Integrating vertical and horizontal partitioning into automated physical database design. In: *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. ACM. pp. 359–370.

Aguilera, Marcos K., Arif Merchant, Mehul Shah, Alistair Veitch and Christos Karamanolis (2009). Sinfonia: A new paradigm for building scalable distributed systems. *ACM Transactions on Computer System*.

Ahamad, Mustaque, Gil Neiger, JamesE. Burns, Prince Kohli and PhillipW. Hutto (1995). Causal memory: definitions, implementation, and programming. *Distributed Computing* **9**(1), 37–49.

Ailamaki, Anastassia, David J. DeWitt, Mark D. Hill and Marios Skounakis (2001). Weaving relations for cache performance. In: *Proceedings of the 27th International Conference on Very Large Data Bases*. pp. 169–180.

Alvisi, Lorenzo, Dahlia Malkhi, Evelyn Pierce and Michael K. Reiter (2001). Fault detection for Byzantine quorum systems. *IEEE Transactions on Parallel Distributed System* **12**(9), 996–1007.

Bailis, Peter, Aaron Davidson et al. (2013*a*). Highly available transactions: virtues and limitations. *Proceedings of the VLDB Endowment* **7**(3), 181–192.

Bailis, Peter, Ali Ghodsi, Joseph M. Hellerstein and Ion Stoica (2013*b*). Bolt-on causal consistency. In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM. pp. 761–772.

Bailis, Peter, Shivaram Venkataraman, Michael J Franklin, Joseph M Hellerstein and Ion Stoica (2012*a*). Probabilistically bounded staleness for practical partial quorums. *Proceedings of the VLDB Endowment* **5**(8), 776–787.

Bailis, Peter, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein and Ion Stoica (2012*b*). Probabilistically bounded staleness for practical partial quorums. *Proceedings of the VLDB Endowment* **5**(8), 776–787.

Baker, Jason, Chris Bond, James Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Léon, Yawei Li, Alexander Lloyd and Vadim Yushprakh (2011). Megastore: Providing scalable, highly available storage for interactive services. In: *In the 5th Conference on Innovative Data Systems Research*. pp. 223–234.

Berenson, Hal, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil and Patrick O'Neil (1995). A critique of ANSI SQL isolation levels. In: *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*. ACM. pp. 1–10.

Bernstein, Philip, Istvan Cseri, Nishant Dani, Nigel Ellis, Ajay Kalhan, Gopal Kakivaya, David B Lomet, Ramesh Manne, Lev Novik, Tomas Talius et al. (2011). Adapting Microsoft SQL server for cloud computing. In: *Proceedings of the 27th International Conference on Data Engineering*. pp. 1255–1263.

Birman, Ken (2007). The promise, and limitations, of gossip protocols. *SIGOPS Operation System Review* **41**(5), 8–13.

Blott, Stephen and Henry F Korth (2002). An almost-serial protocol for transaction execution in main-memory database systems. In: *Proceedings of the 28th International Conference on Very Large Data Bases*. pp. 706–717.

Boncz, Peter, Torsten Grust, Maurice van Keulen, Stefan Manegold, Jan Rittinger and Jens Teubner (2006). MonetDB/XQuery: a fast xquery processor powered by a relational engine. In: *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*. pp. 479–490.

Borkar, Vinayak, Michael J. Carey and Chen Li (2012). Inside "big data management": ogres, onions, or parfaits?. In: *Proceedings of the 15th International Conference on Extending Database Technology*. pp. 3–14.

Bornea, Mihaela A., Orion Hodson, Sameh Elnikety and Alan Fekete (2011). One-copy serializability with snapshot isolation under the hood. In: *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*. pp. 625–636.

Brantner, Matthias, Daniela Florescu, David Graf, Donald Kossmann and Tim Kraska (2008). Building a database on S3. In: *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*. pp. 251–264.

Brewer, Eric A. (2002). Towards robust distributed systems. In: *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing*. ACM. pp. 7–10.

Bridge, William, Ashok Joshi, M. Keihl, Tirthankar Lahiri, Juan Loaiza and N. MacNaughton (1997). The Oracle universal server buffer. In: *Proceedings of the 23rd International Conference on Very Large Data Bases*. pp. 590–594.

Bronson, Nathan, Zach Amsden et al. (2013). Tao: Facebook's distributed data store for the social graph.. In: *USENIX Annual Technical Conference*. pp. 49–60.

Burckhardt, Sebastian, Daan Leijen, Manuel Fähndrich and Mooly Sagiv (2012). Eventually consistent transactions. In: *Proceedings of the 21st European Conference on Programming Languages and Systems*. pp. 67–86.

Burrows, Mike (2006). The chubby lock service for loosely-coupled distributed systems. In: *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*. pp. 335–350.

Cahill, Michael J., Uwe Röhm and Alan D. Fekete (2009). Serializable isolation for snapshot databases. *ACM Transactions on Database Systems* **34**(4), 20:1–20:42.

Campbell, David G., Gopal Kakivaya and Nigel Ellis (2010). Extreme scale with full SQL language support in Microsoft SQL Azure. In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*. pp. 1021–1024.

Chaiken, Ronnie, Bob Jenkins, PerAke Larson, Bill Ramsey, Darren Shakib, Simon Weaver and Jingren Zhou (2008). Scope: easy and efficient parallel processing of massive data sets. *Proceedings of the VLDB Endowment* **1**(2), 1265–1276.

Chandra, Tushar D., Robert Griesemer and Joshua Redstone (2007). Paxos made live: an engineering perspective. In: *Proceedings of the 26th Annual ACM Symposium on Principles of Distributed Computing*. pp. 398–407.

Chang, Fay, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes and Robert E. Gruber (2008). Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems* **26**(2), 4:1–4:26.

Cipar, James, Greg Ganger, Kimberly Keeton, Charles B. Morrey, III, Craig A.N. Soules and Alistair Veitch (2012). Lazybase: trading freshness for performance in a scalable database. In: *Proceedings of the 7th ACM European Conference on Computer Systems*. pp. 169–182.

Cohen, Jeffrey, Brian Dolan, Mark Dunlap, Joseph M Hellerstein and Caleb Welton (2009). Mad skills: new analysis practices for big data. *Proceedings of the VLDB Endowment* **2**(2), 1481–1492.

Condie, Tyson, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, John Gerth, Justin Talbot, Khaled Elmeleegy and Russell Sears (2010). Online aggregation and continuous query support in MapReduce. In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*. pp. 1115–1118.

Cooper, Brian F, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan and Russell Sears (2010). Benchmarking cloud serving systems with YCSB. In: *Proceedings of the 1st ACM Symposium on Cloud computing*. ACM. pp. 143–154.

Cooper, Brian F., Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver and Ramana Yerneni (2008). PNUTS: Yahoo!'s hosted data serving platform. Vol. 1. pp. 1277–1288.

Corbett, James C., Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang and Dale Woodford (2012). Spanner: Google's globally-distributed database. In: *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*. pp. 251–264.

Curino, Carlo, Evan Jones, Yang Zhang and Sam Madden (2010). Schism: a workload-driven approach to database replication and partitioning. *Proceedings of the VLDB Endowment* **3**(1-2), 48–57.

Dean, Jeffrey and Sanjay Ghemawat (2008). MapReduce: Simplified data processing on large clusters. *Communication of ACM* **51**(1), 107–113.

DeCandia, Giuseppe, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels (2007). Dynamo: Amazon's highly available key-value store. In: *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles.* ACM. pp. 205–220.

Défago, Xavier, André Schiper and Péter Urbán (2004). Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computer Survey.*

Escriva, Robert, Bernard Wong and Emin Gün Sirer (2012). Hyperdex: a distributed, searchable key-value store. *SIGCOMM Computer Communication Review* **42**(4), 25–36.

Fekete, Alan, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil and Dennis Shasha (2005). Making snapshot isolation serializable. *ACM Transactions on Database Systems* **30**(2), 492–528.

Geilen, Marc and Twan Basten (2003). Requirements on the execution of Kahn process networks. Springer. pp. 319–334.

Ghemawat, Sanjay, Howard Gobioff and Shun-Tak Leung (2003). The Google file system. In: *Proceedings of the 19th ACM Symposium on Operating Systems Principles.* ACM. pp. 29–43.

Gibson, Garth A and Rodney Van Meter (2000). Network attached storage architecture. *Communications of the ACM* **43**(11), 37–45.

Gilbert, Seth and Nancy Lynch (2002). Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News* **33**(2), 51–59.

Gligor, Gabriela and Silviu Teodoru (2011). Oracle exalytics: Engineered for speed-of-thought analytics. *Database Systems Journal* **2**(4), 3–8.

Gribble, Steven David (2000). A Design Framework and a Scalable Storage Platform to Simplify Internet Service Construction. PhD thesis. University of California, Berkeley. AAI3001853.

Grund, Martin, Jens Krüger, Hasso Plattner, Alexander Zeier, Philippe Cudre-Mauroux and Samuel Madden (2010). Hyrise: a main memory hybrid storage engine. *Proceedings of the VLDB Endowment* **4**(2), 105–116.

Gummadi, K., R. Gummadi, S. Gribble, S. Ratnasamy, S. Shenker and I. Stoica (2003). The impact of DHT routing geometry on resilience and proximity. In: *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications.* pp. 381–394.

Hankins, Richard A. and Jignesh M. Patel (2003). Data morphing: an adaptive, cache-conscious storage technique. In: *Proceedings of the 29th International Conference on Very Large Data Bases.* pp. 417–428.

Harizopoulos, Stavros and Anastassia Ailamaki (2003). A case for staged database systems. In: *CIDR.* number DIAS-CONF-2003-001.

Helland, Pat (2007). Life beyond distributed transactions: an apostate's opinion. In: *In 3rd Biennial Conference on Innovative Data Systems Research.* pp. 132–141.

Hochschild, Peter, Alexander Lloyd, Wilson Cheng-Yi Hsieh, Robert Edman Felderman and Michael James Boyer Epstein (2015). Generating globally coherent timestamps. US Patent 8,938,636.

Hunt, Patrick, Mahadev Konar, Flavio P. Junqueira and Benjamin Reed (2010). Zookeeper: wait-free coordination for internet-scale systems. In: *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*. pp. 11–11.

Isard, Michael, Mihai Budiu, Yuan Yu, Andrew Birrell and Dennis Fetterly (2007). Dryad: distributed data-parallel programs from sequential building blocks. *ACM SIGOPS Operating Systems Review* **41**(3), 59–72.

Johnson, Ryan, Ippokratis Pandis, Nikos Hardavellas, Anastasia Ailamaki and Babak Falsafi (2009). Shore-MT: a scalable storage manager for the multicore era. In: *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*. pp. 24–35.

Jones, Evan P.C., Daniel J. Abadi and Samuel Madden (2010). Low overhead concurrency control for partitioned main memory databases. In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*. pp. 603–614.

Jorwekar, Sudhir, Alan Fekete, Krithi Ramamritham and S. Sudarshan (2007). Automating the detection of snapshot isolation anomalies. In: *Proceedings of the 33rd International Conference on Very Large Data Bases*. pp. 1263–1274.

Joshi, Ashok, Sam Haradhvala and Charles Lamb (2012). Oracle NoSQL database-scalable, transactional key-value store. In: *Proceedings of the 2nd International Conference on Advances in Information Mining and Management*. pp. 75–78.

Kahn, Gilles (1974). The semantics of a simple language for parallel programming. North-Holland.

Kallman, Robert, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg and Daniel J. Abadi (2008). H-store: a high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment* **1**(2), 1496–1499.

Karger, David, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine and Daniel Lewin (1997). Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In: *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*. ACM. pp. 654–663.

Kemme, Bettina and Gustavo Alonso (2000). A new approach to developing and implementing eager database replication protocols. *ACM Transactions on Database Systems* **25**(3), 333–379.

Kossmann, Donald, Tim Kraska and Simon Loesing (2010). An evaluation of alternative architectures for transaction processing in the cloud. In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*. ACM. pp. 579–590.

Kraska, Tim, Gene Pang, Michael J. Franklin, Samuel Madden and Alan Fekete (2013). MDCC: multi-data center consistency. In: *Proceedings of the 8th ACM European Conference on Computer Systems*. pp. 113–126.

Lahiri, Tirthankar, Vinay Srihari, Wilson Chan, N. MacNaughton and Sashikanth Chandrasekaran (2001). Cache fusion: Extending shared-disk clusters with shared caches. In: *Proceedings of the 27th International Conference on Very Large Data Bases*. pp. 683–686.

Lakshman, Avinash and Prashant Malik (2010). Cassandra: a decentralized structured storage system. *SIGOPS Operation System Review* **44**(2), 35–40.

Larson, Per-Åke, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M Patel and Mike Zwilling (2011). High-performance concurrency control mechanisms for main-memory databases. *Proceedings of the VLDB Endowment* **5**(4), 298–309.

Lee, Edward A. (2011). Concurrent models of computation: An actor-oriented approach. Ptolemy.org. UC Berkeley. pp. 17–26.

Lee, Edward A and Thomas M Parks (1995). Dataflow process networks. Vol. 83. pp. 773–801.

Lewis, Philip M, Arthur J Bernstein and Michael Kifer (2002). *Databases and transaction processing: an application-oriented approach.* Addison-Wesley.

Lin, Yi, Bettina Kemme, Marta Patiño Martínez and Ricardo Jiménez-Peris (2005). Middleware based data replication providing snapshot isolation. In: *Proceedings of the 2005 ACM SIGMOD international conference on Management of data.* pp. 419–430.

Lloyd, Wyatt, Michael J. Freedman, Michael Kaminsky and David G. Andersen (2011). Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In: *Proceedings of the 23rd ACM Symposium on Operating Systems Principles.* pp. 401–416.

Lloyd, Wyatt, Michael J. Freedman, Michael Kaminsky and David G. Andersen (2013). Stronger semantics for low-latency geo-replicated storage. In: *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation.* pp. 313–328.

Lynch, Nancy A. and Eugene W. Stark (1989). A proof of the Kahn principle for input/output automata. *Information and Computation* **82**(1), 81 – 92.

Manassiev, Kaloian, Madalin Mihailescu and Cristiana Amza (2006). Exploiting distributed version concurrency in a transactional memory cluster. In: *Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* pp. 198–208.

Manegold, Stefan, Martin L. Kersten and Peter Boncz (2009). Database architecture evolution: mammals flourished long before dinosaurs became extinct. *Proceedings of the VLDB Endowment* **2**(2), 1648–1653.

Melnik, Sergey, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton and Theo Vassilakis (2010). Dremel: interactive analysis of web-scale datasets. *Proceedings of the VLDB Endowment* **3**(1-2), 330–339.

Olston, Christopher, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar and Andrew Tomkins (2008). Pig latin: a not-so-foreign language for data processing. In: *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data.* pp. 1099–1110.

Özsu, M Tamer and Patrick Valduriez (2011). *Principles of distributed database systems.* Springer Science & Business Media.

Patterson, Stacy, Aaron J. Elmore, Faisal Nawab, Divyakant Agrawal and Amr El Abbadi (2012). Serializability, not serial: concurrency control and availability in multi-datacenter datastores. *Proceedings of the VLDB Endowment* **5**(11), 1459–1470.

Poess, Meikel and Raghunath Othayoth Nambiar (2005). Large scale data warehouses on grid: Oracle database 10g and HP proliant servers. In: *Proceedings of the 31st International Conference on Very Large Data Bases.* pp. 1055–1066.

Pritchett, Dan (2008). BASE: An ACID alternative. *Queue* **6**(3), 48–55.

Qiao, Lin, Kapil Surlaker, Shirshanka Das, Tom Quiggle, Bob Schulman, Bhaskar Ghosh, Antony Curtis, Oliver Seeliger, Zhen Zhang, Aditya Auradar et al. (2013). On brewing fresh Espresso: Linkedin's distributed data serving platform. In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. pp. 1135–1146.

Ramamurthy, Ravishankar, David J. DeWitt and Qi Su (2003). A case for fractured mirrors. *The VLDB Journal* **12**(2), 89–101.

Rao, Jun, Eugene J. Shekita and Sandeep Tata (2011). Using paxos to build a scalable, consistent, and highly available datastore. *Proceedings of the VLDB Endowment* **4**(4), 243–254.

Roh, Hyun-Gul, Myeongjae Jeon, Jin-Soo Kim and Joonwon Lee (2011). Replicated abstract data types: Building blocks for collaborative applications. *Journal of Parallel Distributed Computer* **71**(3), 354–368.

Saito, Yasushi and Marc Shapiro (2005). Optimistic replication. *ACM Computer Survey* **37**(1), 42–81.

Schiper, Nicolas, Pierre Sutra et al. (2010). P-store: Genuine partial replication in wide area networks. In: *The 29th IEEE Symposium on Reliable Distributed Systems*. pp. 214–224.

Shapiro, Marc, Nuno Preguiça, Carlos Baquero and Marek Zawirski (2011). Conflict-free replicated data types. In: *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*. pp. 386–400.

Shute, Jeff, Radek Vingralek, Bart Samwel, Ben Handy, Chad Whipkey, Eric Rollins, Mircea Oancea, Kyle Littlefield, David Menestrina, Stephan Ellner et al. (2013). F1: A distributed SQL database that scales. *Proceedings of the VLDB Endowment* **6**(11), 1068–1079.

Shvachko, Konstantin, Hairong Kuang, Sanjay Radia and Robert Chansler (2010). The Hadoop distributed file system. In: *Proceedings of the 26th Symposium on Mass Storage Systems and Technologies*. pp. 1–10.

Sikka, Vishal, Franz Färber, Wolfgang Lehner, Sang Kyun Cha, Thomas Peh and Christof Bornhövd (2012). Efficient transaction processing in SAP HANA database: the end of a column store myth. In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. pp. 731–742.

Stonebraker, Michael (2012). New opportunities for NewSQL. *Communications of the ACM* **55**(11), 10–11.

Stonebraker, Michael, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem and Pat Helland (2007). The end of an architectural era: (it's time for a complete rewrite). In: *Proceedings of the 33rd International Conference on Very Large Data Bases*. pp. 1150–1160.

Stonebraker, Mike, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran and Stan Zdonik (2005). C-store: a column-oriented dbms. In: *Proceedings of the 31st International Conference on Very Large Data Bases*. pp. 553–564.

Terry, Doug (2011). Replicated data consistency explained through baseball. Technical report. Technical Report MSR-TR-2011-137, Microsoft Research.

Thomas, R.H (1984). A solution to the concurrency control problem for multiple copy databases. In: *Digest of papers IEEE COMPCON Spring*. pp. 56–62.

Thomson, Alexander, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao and Daniel J Abadi (2012). Calvin: Fast distributed transactions for partitioned database systems. In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. pp. 1–12.

Thusoo, Ashish, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff and Raghotham Murthy (2009). Hive: a warehousing solution over a MapReduce framework. *Proceedings of the VLDB Endowment* **2**(2), 1626–1629.

Thusoo, Ashish, Zheng Shao, Suresh Anthony, Dhruba Borthakur, Namit Jain, Joydeep Sen Sarma, Raghotham Murthy and Hao Liu (2010). Data warehousing and analytics infrastructure at Facebook. In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*. pp. 1013–1020.

Verma, Abhishek, Brian Cho, Nicolas Zea, Indranil Gupta and Roy H. Campbell (2013). Breaking the MapReduce stage barrier. *Cluster Computing* **16**(1), 191–206.

Vogels, Werner (2008). Eventually consistent. *Queue* **6**(6), 14–19.

Weikum, Gerhard and Gottfried Vossen (2001). *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery*. Elsevier.

Welsh, Matt, David Culler and Eric Brewer (2001). SEDA: an architecture for well-conditioned, scalable internet services. In: *Proceedings of the 18th ACM Symposium on Operating Systems Principles*. pp. 230–243.

Wu, Lengdong, Hua He and Osmar R Zaïane (2013). Utility enhancement for privacy preserving health data publishing. In: *Advanced Data Mining and Applications*. pp. 311–322.

Wu, Lengdong, Liyan Yuan and Jiahuai You (2014). BASIC, an alternative to BASE for large-scale data management system. In: *Proceedings of the 2014 IEEE International Conference on Big Data*. pp. 5–14.

Wu, Lengdong, Liyan Yuan and Jiahuai You (2015). Survey of large-scale data management systems for big data applications. *Journal of computer science and technology* **30**(1), 163–183.

Wu, Shuqing and Bettina Kemme (2005). Postgres-R (SI): Combining replica control with concurrency control based on snapshot isolation. In: *Proceedings of the 21st International Conference on Data Engineering*. pp. 422–433.

Xin, Reynold S, Josh Rosen, Matei Zaharia, Michael J Franklin, Scott Shenker and Ion Stoica (2013). Shark: SQL and rich analytics at scale. In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. pp. 13–24.

Xu, Yu, Pekka Kostamaa, Xin Zhou and Liang Chen (2008). Handling data skew in parallel joins in shared-nothing systems. In: *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*. pp. 1043–1052.

Yu, Haifeng and Amin Vahdat (2002). Minimal replication cost for availability. In: *Proceedings of the 21st Annual Symposium on Principles of Distributed Computing*. pp. 98–107.

Yu, Haifeng and Amin Vahdat (2006). The costs and limits of availability for replicated services. *ACM Transactions on Computer Systems* **24**(1), 70–113.

Yuan, Li-Yan, Lengdong Wu, Jia-Huai You and Yan Chi (2015). A demonstration of Rubato DB: A highly scalable NewSQL database system for OLTP and big data applications. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM. pp. 907–912.

Yuan, Liyan, Lengdong Wu, Jiahuai You and Yan Chi (2014). Rubato DB: a highly scalable staged grid database system for OLTP and big data applications. In: *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*. ACM. pp. 1–10.

Zaharia, Matei, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker and Ion Stoica (2010). Spark: cluster computing with working sets. In: *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*. Vol. 10. pp. 1–7.