Trading Consistency for Synchronization Cost Reduction in Real-time On-Line Analytical Processing (OLAP) Systems

by

You Li

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

Master of Science

Department of Computing Science
University of Alberta

Abstract

A real-time OLAP system caches previous queries' results to accelerate the processing of future queries. On such a system, whenever an update happens, affected cached values must be invalidated, or re-calculated, to maintain their consistency. Several different invalidation policies are used in real-time OLAP system. For all these policies, during the invalidation, the system must perform synchronization to ensure the consistency of cached values. Such synchronization can lead to poor scalability and may, therefore, dramatically degrades the throughput of the OLAP system.

A synchronization-free invalidation policy was introduced in previous work to improve scalability. Such a policy results in stale cached values that can lead to incorrect query answers. To reduce the level of inconsistency in the system, this policy relies on independent threads, called *fresheners*, to recompute stale values. This thesis evaluates this policy with a manufacturer data set. It introduces new metrics to measure the level of inconsistency in the system. Based on these metrics, it measures the throughput improvement and number of incorrect results generated with this policy.

Invalidation incurs high overhead even when the system is not performing synchronization. An alternative is to skip invalidation and to rely solely on fresheners to update incorrect results. This thesis introduces such a policy. This policy requires a more sophisticated strategy to keep the number of stale cached values to an acceptable level.

This thesis also develops a framework to monitor the current status of the OLAP system and adjusts the number of fresheners. The goal of this framework is to deliver a stable, and acceptable, probability of incorrect results, due to stale cached values, under various workload.

Acknowledgements

Foremost, I would like to express my sincere gratitude to my advisor Jose Nelson Amaral for the continuous support of my Master study and research, for his patience, motivation, enthusiasm, and immense knowledge. His guidance helped me throughout the research and the writing of this thesis. I cannot imagine having a better advisor and mentor for my Master study.

My sincere thanks also goes to David Ungar and Doug Kimelman who pioneered the synchronization-free approach for OLAP system and offered numerous important suggestions for my research.

I thank Jeeva Paudel, Matthew Gaudet and all members in Nelson's team for their help with my project. I am also glad to share time on the Compiler Design and Optimization Laboratory (CDOL) seminars with them reading and discussing published academic papers.

I gratefully acknowledge the funding received towards my Master from the Alberta Innovates Technology Futures and support from the IBM Alberta Centre for Advanced Studies. I would like to thank the IBM Canada Software Laboratory, especially Adam D'Andrea, who granted me access to the POWER machine to do the experiments and helped me deal with related issues.

Lastly, I would like to thank my family for their years of support, without which this work would not have been possible.

Table of Contents

List of Tables

List of Figures

List of Acronyms

OLAP On-line Analytical Processing

ROLAP Relational On-line Analytical Processing

MOLAP Multidimensional On-line Analytical Processing

NC No Cache

SI Synchronized Invalidation

SCC Synchronized Cache Clearing

UI Unsynchronized Invalidation

NINS No Invalidation and No Synchronization

ACB MRQ Access-Counter-Based MRQ

TB MRQ Time-Based MRQ

Chapter 1

Introduction

Caching is widely used in query systems to accelerate the response time of future queries by reusing the values produced for previous queries. Invalidating cached values when an update occurs in data that was used to compute such values is essential to ensure the correct operation of the query system. Unfortunately, the invalidation procedure is non-trivial on some system and can incur significant overhead. This thesis investigates several widely accepted invalidation policies on real-time on-line analytical processing (OALP) systems. Given the limitations in existing policies, we propose an innovative policy that eliminates invalidation during update. The elimination of invalidation leads to incorrectly cached values and the policy relies on separate threads to fix the cached values. However, some incorrect cached values may not be fixed in a timely manner, and thus the user may still get incorrect answers for queries calculated using cached value. A new framework that manages the tradeoff between the number of fresheners used in the system and the level of incorrectness that is tolerated while adapting to various workload.

Business intelligence(BI) systems are an essential tool to support decision making and play a crucial role in many organizations. Such systems have evolved very significantly since they were first introduced.

Traditionally, an OLAP system, which is an important components of BI systems, works based on a data mirror that is usually extracted from on-line transaction processing (OLTP) systems. Limited by the time needed, the extraction procedure can only be performed periodically instead of on demand. Periodical extraction of data from an OLTP system leads the OLAP system to answer user's query based on stale data. The time lag between the data extraction and the OLAP queries can become an obstacle to reacting to changes in an organization in a timely fashion.

Real-time OLAP systems are becoming more broadly used [31]. White points out the need for real-time OLAP systems that allow users to make decision based on almost-current information [31]. On such system, the OLAP part can no longer be separated from the OLTP part because the time required to extract data from the OLTP system leads to stale OLAP query answers. In other words, a real-time OLAP system must include features that traditionally exist only in OLTP systems (*e.g.*, update on data). Significant effort has gone into the design of integrated OLAP and OLTP system with acceptable performance on both types of operations [26]. Real-time OLAP systems must concurrently answer complex queries and execute updates into very large data-set.

Individual queries can be quite complex and time consuming. Therefore cache systems that record previous answered queries' results are implemented to improve the responding time and throughput of some current real-time OLAP systems such as the IBM cognos TM1. Including OLTP features (*e.g.*, update operation) in such systems requires that the designer deal with the cache-invalidation issue. Then, a crucial question must be investigated: will benefit of cache system, under certain workload, be sufficient to pay for overhead of the invalidation procedure? The cache system should work well on a system without frequent updates. However, is there an invalidation policy that make the cache system advantageous on various types of workload?

Research effort has been directed toward lowering the overhead on invalidation, especially on many-thread system. Ungar *et al.* found that the synchronization during invalidation generates larger mount of overhead and that this overhead increases with the number of threads the system uses[30]. They proposed to eliminate the synchronization during invalidation and found throughput benefits on small-size data sets [30]. Then how would their synchronization-free approach performs on larger data set with much more complex queries?

Ungar *et al.* propose that inconsistency could be tolerated in some OLAP applications. An interesting research question is whether it is possible to go one step further and eliminate the whole invalidation procedure instead of just the synchronization part? This paper implements such invalidation-free policy for the update procedure and compares its performance with exist policies.

Both synchronization-free policies and invalidation-free policies lead to inconsistency. Thus, it is important to have metrics to measure this inconsistency. Also,

an important question is how bad the inconsistency issue becomes and can it be somehow controlled? The thesis proposed metrics to measure the level of inconsistency in the system. The thesis also introduces a framework to adjust the amount of resources dedicated to freshening stale cached cells, thus affecting the level of inconsistency in the OLAP system.

This dissertation aims to provide support to the following thesis statement:

> The elimination of invalidation during cell update, on real-time OLAP systems, may provide significant performance benefits at the cost of producing inconsistency results for queries.

This thesis is introduced based on assumption that, in some application domains for real-time OLAP, come level of inconsistency is tolerable.

To contribute to the state of the art of real-time OLAP systems this thesis:

- Compares the throughput of synchronized invalidation policies with the throughput of the synchronization-free invalidation policy. Then it measures the inconsistency level for Ungar's synchronization-free approach [30] .
- Proposes a new invalidation-free policy for that is based on the approach proposed by Ungar [30]. Independent threads, named *freshener*, keep inconsistency at an acceptable level. The number of fresheners used in the system affect the cached-value stale rate.
- Proposes the Adaptive Inconsistency Level Framework (AILF) to monitor the inconsistency level dynamically and to adapt the number of fresheners to the current workload.
- Builds a real-time in-memory OLAP system prototype. This prototype is able to load in data from an IBM Cognos TM1's dumped file. It builds a data cube according to the dimension hierarchy and TM1 rules. TM1 rules allows a cell in a data cube to be calculated from any other cells using an arbitrary equation. This prototype includes the cache system that stores previous queries' results to be used in future calculations.

The rest of the thesis is organized as follows. Chapter 2 provides essential background knowledge and reviews related work. Chapter 3 describes the synchronization policies and motivates the need for synchronization to maintain the consistency

of cached values. Chapter 4 introduces Ungar's synchronization-free policy and our new invalidation-free policy and presents a new freshening strategy, most-recent-query(MRQ) freshening, to meet the requirement on efficiency in an invalidation-free policy. Chapter 5 proposes the AILF that adaptively adjusts the number of fresheners to maintain an acceptable possibility of generating incorrect results while adapting to various workload. Chapter 7 concludes this thesis and describes future work.

Chapter 2

Background

This chapter starts with a brief history of the evolution of decision-support systems (DSS). Then it presents an overview of On-Line Analytical Processing (OLAP) and On-Line Transaction Processing (OLTP) applications. One of the most important concepts to understand OLAP is the structure of a data-cube model, including relation of the values stored in multiple cells and the mechanism used to store and update the value of the cells. Thus, the chapter presents the necessary background to understand the data cube model. The chapter ends with related research in the areas of OLAP, approximate computing, and incremental computation.

2.1 Decision-Support Systems

Decision-Support Systems (DSS) became essential tools for the management of data and for decision making in large corporations [23]. The development of the IBM System 360 and other "powerful" mainframe systems made it realistic to build Management Information Systems (MIS), which were considered very expensive prior to 1965. With data from accounting and transaction system, MIS is able to provide managers with structured, periodic, reports for decision making.

In 1971, a seminal book by McKenney and Scott introduces the idea of DSS [19]. The development of the theory to support DSS starts in the late 1970s. For instance, Sprague and Carlson's seminal book is regarded as a crucial milestone in the history of DSS because it provides practical guidance to build a DSS [28].

Beginning in about 1990, data warehousing and On-Line Analytical Processing (OLAP), which can be categorized as Data-Driven DSS, gain popularity [23]. At the time, Bill Inmon, who is often referred to as "the father of data warehouse" defined DSS as "a system used to support managerial decisions." A key aspect of

DSS processing — which distinguishes it from the on-line systems that came later — is that a DSS does not provide support for the update of the data set. In other words, DSS is only concerned with providing analytic results for a given static data set. DSS does not support collecting or updating data and therefore has no control on the quality of the data used for the analysis.

2.2 OLAP and OLTP applications

On-Line Transaction Processing (OLTP) applications deal with tasks that are essential for the daily operation of an organization such as recording order entry and banking transactions. These tasks are structured and repetitive and they are executed through a series of short transactions that are both atomic and isolated [9]. Each transaction must be recorded and updated on time. For large organizations a huge amount of transactions must be processed simultaneously. Therefore consistency, recoverability and high throughput are key requirements for the design of databases to support OLTP applications.

As an essential tool for decision support system, On-Line Analytical Processing (OLAP) applications aim to help managers make decisions. While the focus of OLTP applications is fast throughput in the processing of detailed individual records, the focus of OLAP applications is historical, summarized and consolidated, data. Therefore, a typical OLAP query requires much more data access and computation compared to an OLTP query. For example, a manager in Audi, a car manufacturer, is more likely to need the answers for queries such as how many cars the company has sold last year and which car model sold best last month than the details of an specific car-sale transaction recorded by an OLTP application. Such complex queries may be related with a large number of individual transactions selected by specific features (*e.g.* model, year or color). Data-driven decision making leads to more sophisticated OLAP queries such as "what is the best-selling car for buyers between twenty and thirty year olds in cold climate?" In general, each query in an OLAP application is non-trivial to answer, but both the response time to individual queries and the throughput processing of queries are very important criteria to measure the performance of an OLAP application. In summary, the aim of an OLAP system is to provide fast answers to multi-dimensional analytical (MDA) queries [12].

Traditionally, OLAP applications are maintained separately from an organiza-

tion's operating database that is designed to support (OLTP) applications. The difference of workload and requirements between OLTP and OLAP applications makes it difficult for a single database to support both of them with acceptable performance. A concrete example should illustrate the optimization dilemma for a database intended to support both OLTP and OLAP. To answer an OLAP query about the sales for a specific model (*e.g.* A4 of Audi), the database either needs to build an index on column "model" or it has to iterate over all records. Unfortunately, building indexes on all features that could possible appear in an OLAP query results in significant overhead on insertion and deletion into the database. Insertions and deletions are frequently executed by OLTP applications.

Therefore OLAP applications usually work on separate databases that are optimized for OLAP queries' requirements. OLAP applications rely on an Extract-Transform-Load (ETL) tool to extract data from OLTP applications' operating databases and then re-format the data to an structure that is optimized for OLAP applications. This solution has been widely accepted by the community and applied by industry since early 1990s when data warehousing and OLAP were defined.

Using separate databases for OLAP applications leads to the data freshness problem. The ETL procedure is time consuming and therefore can only be executed periodically (seasonal, monthly or weekly). Therefore, OLAP applications that rely only on ETL would answer user queries with out-of-date data and thus generate reports with stale results. Early in the history of OLAP such stale results were regarded to be acceptable because it was the only practical solution for the technology of the time.

However, with the support of current technology that with much more powerful processors and more memory space than the machines of the IBM System 360 vintage, it is no longer an unrealistic goal to build a "real-time" OLAP application that generates results based on data that is very close to up-to-date. IBM Cognos TM1 and icCube are the pioneers of real-time OLAP applications [3, 4]. A real-time OLAP application is either an integrated OLAP and OLAP system which is able to deal with OLAP transactions by itself to keep the data freshness or implemented with efficient tool to extract data from OLTP system on real-time. Both the transaction processing and the extraction brings overhead to the OLAP application. The design decision for both the IBM Cognos TM1 and icCube solutions is to store all the operating data in memory to reduce the overhead incurred for real-time

processing. The in-memory approach makes the responding time and throughput acceptable but imposes constraints on the size of the data.

On traditional OLAP applications, pre-computation on all the possible queries is a popular strategy to accelerate the query response time because each individual query could be quite complex and thus time consuming. Such strategy only makes sense if the whole dataset needed to answer a query remains stable for a period of time. Therefore this strategy is not very useful for a real-time OLAP application where the data changes continuously. Fortunately, even though any portion of the dataset may be changed at any time, some of the query answers remain stable over a period of time. A query previously answered whose data has not changed should not be re-calculated when issued a second time. For this case, IBM Cognos TM1 implements a cache system that stores prior queries' answers to accelerate future query's response time.

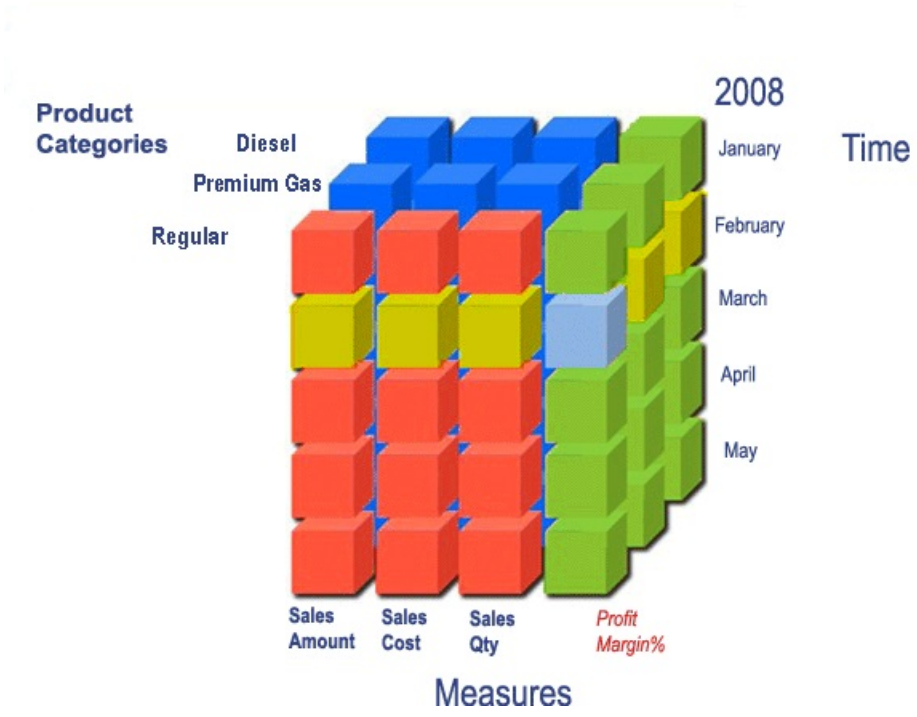
2.3 Data cube

In OLAP, a high-dimensional hierarchical *cube* is used as an abstraction to present and discuss the organization of the data. This cube can be understood as a n-dimensional generalization of a two-dimensional spreadsheet, but with the added capacity of aggregating multiple points in a dimension, which are equivalent to rows or columns in a two dimensional structure, into aggregated coordinate. For instance, January, February, and March may be aggregated into a First Quarter coordinate.

2.3.1 Cube Overview

OLAP uses a *multi-dimensional dataset*, often called a *data cube*, as its logical model. A data cube is an array of data understood in terms of its *dimensions*. Figure 2.1, reproduced from website of Datanova[2], depicts a simple data cube that consists of 3 dimensions: Time dimension on vertical axis, Measure dimension on horizontal axis and Product Category dimension on the depth axis [2]. The members in each dimension (*e.g.* January in Time dimension) are named coordinates. A *coordinate-tuple* consists of n coordinates, one from each dimension of the n-dimensional data cube and it uniquely identifies a *cell*. Conceptually, each cell of the cube holds a value. Some cells hold the value entered by a user and are thus called *entered cells*. Other cells are calculated from those entered ones and are thus called *computed cells*. The cube's persistence storage may not contain the computed cells because

Figure 2.1: Simple OLAP Cube (reproduced from [2])



the values of them may always be calculated on the fly.

2.3.2 Hierarchical Dimension

The dimensions can be much more complex compared to the simple ones in Figure 2.1 because there can be a hierarchical relation between coordinates within a single dimension. Figure 2.2 shows a data cube with hierarchical dimensions and Figure 2.3 shows the hierarchical relation among Time dimension. Figures 2.2 and 2.3 are reproduced from a blog post by Jagadish Chaterjee in ASP/free, an online community focused on the Microsoft web framework [1]. As shown in Figure 2.3, the coordinates in a hierarchical dimension are organized as directed acyclic graph (DAG). It is clear that January 1st is the child coordinate of January, which has parent coordinate Quarter 1, and grandparent coordinate 1990. The hierarchical relation among coordinates is usually, but not necessarily, designed with logical meaning. This relation specifies that the cell with non-leaf coordinate can be calculated from the cells with the child coordinates. For instance, for the cube in Figure 2.2, if we take the cell at South America for Route dimension, 1st half in Time dimension and air in Source dimension — such coordinate is denoted as [South America, 1st half, air] — the query *aggregation*(sum) of [South America, 1st quar-

ter, air] returns 600 and [South America, 2nd quarter, air] returns 490 which equals 1090, because of the hierarchical relation between 1st quarter, 2nd quarter and 1st half. Sum is not the only operation that can be applied to calculate the value of a cell from its child cell's values. Any other aggregation operation — for instance average or standard deviation — can be computed.

Cell Hierarchy: Let cells A and B be two cells that belong to the same cube $\mathcal{C}$. Cell A is an *ascendent* of cell B(or Cell B is an *descendent* of cell A) if and only if:

1. The coordinates of A on each dimension of $\mathcal{C}$ are ascendents of or same as the corresponding coordinates of B, and

2. at least one of the coordinates of A is an ascendent of a corresponding coordinate of B.

Given cells A and B, both belonging to a cube $\mathcal{C}$, cell A is a *parent* cell of B, or B is a *child* of A, if and only if:

1. There exist a dimension in which the coordinate of A is a parent of the coordinate of B.

2. A and B have exactly same coordinate in all other dimensions.

With hierarchical dimensions, the cells can be categorized into leaf cells and consolidated cells. A cell is a leaf cell if none of its coordinates has descendents, otherwise it is a consolidated cell. A consolidated cell is one kind of computed cell – computed by consolidation.

2.3.3 Rules

Most data-cube models only allow a user to define relations among cells within a dimension hierarchy as discussed previously in Section 2.3.2. However, TM1 introduces a more flexible way to defined cell relations. In TM1, besides aggregation from child cells, a computed cell may also be calculated following pre-defined *rules*. A rule defines how a group of cells that satisfies certain constraints should be calculated. For example, the following rule can be applied to the cube in Figure 2.1.

$$[\mathit{ProfitMargin}\%] \quad = \quad N:([\text{SalesAmount}] - [\text{SalesCost}])/[\text{SalesQty}]$$

Figure 2.2: Cube with hierarchical dimensions (reproduced from [1])

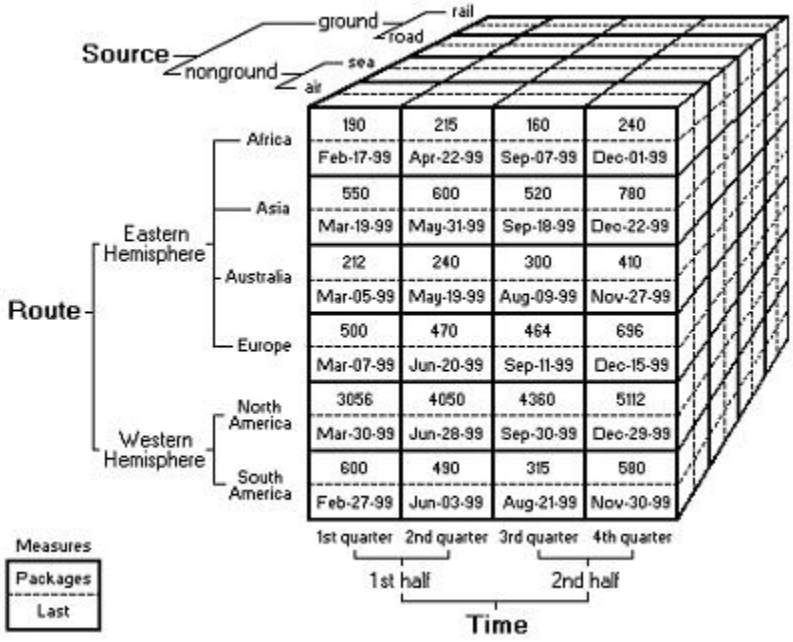
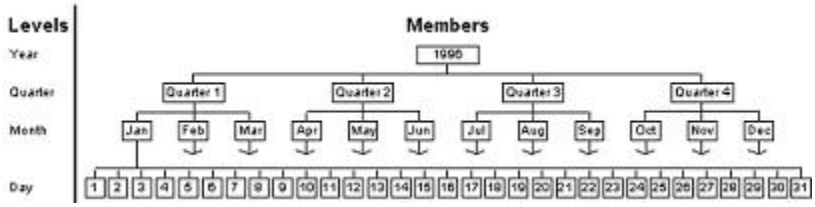


Figure 2.3: Time Dimension (reproduced from [1])



The left part of the rule states that it applies to cells on the *ProfitMargin%* coordinate in the Measure dimension. The N : on the right part makes the rule only apply to leaf cells. Thus, this rule results on the following equation:

$$[R, J, \textit{ProfitMargin}\%] = \frac{([R, J, \text{SalesAmount}] - [R, J, \text{SalesCost}])}{[R, J, \text{SalesQty}]} \tag{2.1}$$

Where R stands for Regular and J stands for January. This rule also applies to cells with all different leaf coordinate in *category* and *time* dimension.

2.3.4 Cached Cell and Staleness

A lazy computation strategy is adopted by real-time OLAP applications such as the IBM Cognos TM1. The system does not calculate the value of a computed cell until such value is queried. To reduce the response time and improve the throughput, once a computed cell value is calculated that value is cached to answer future queries. A computed cell that holds a cached value from prior queries is called a *cached cell*. Intuitively, the cached value will be stale (out-of-date) if it is *dependent* on a cell whose value has changed after the cached value was computed. A cached cell is also stale if a new cell that it depends upon has been inserted into the data cube.

Cell Dependency: Cell A *depends* on Cell B if

1. cell B is a descendent cell of A; or
2. there exist a rule r that applies on A and use the value of B to calculate the value of A; or
3. there exist a rule r that applies on A and use the value of a cell D to calculate A and D is depends on B.

Cell A *directly depends* on cell B if

- cell B is child cell of A; or
- there exist a rule r that applies on A and use the value of B to calculate A.

On TM1, the value of a empty entered cell (i.e. the entered cells has not been updated) is regarded to be zero. Therefore, the insertion and deletion of entered cells can be regarded as special cases of value update of the entered cell. In general a cached cell A is *stale* if cell A depends on a cell B and the value of cell B has

changed after the cached value of cell A was calculated. The value of a cell may change through recalculation, data entry, cell deletion, or cell insertion.

A OLAP application running on a system that caches calculated values must either have a mechanism to detect and refresh stale values or it must tolerate stale answer to queries. Chapters 3 and 4 introduce existing policies to maintain the correctness of cached values during update and policies that require the application to tolerate stale cached values.

2.4 Related Work

The investigation presented in this thesis is related to previous work in the areas of OLAP, approximate computing, and incremental computation.

2.4.1 OLAP

Throughput and latency are two key metrics to measure OLAP application's performance. Research on different aspect of OLAP and data warehousing has been focused on the improvement of throughput and on the reduction of latency.

The design of efficient data structures to store data on disk is an important focus of previous research on OLAP. Such research often investigates different specialized multidimensional data structures, leading to what is called multidimensional OLAP (MOLAP) [22]. Zhao *et al.* propose variety optimization on multidimensional arrays to handle sparse arrays and to improve performance [29, 32]. Relational OLAP (ROLAP) applications use a relational database source and rely on tools to access data in that database through SQL queries that calculate the information requested by the user. Compared to OLTP applications, which are also based on relational database, ROLAP employs specialized index structures to achieve good performance. Substantial research effort has been directed toward optimizing relational databases for OLAP workloads [17, 14, 13, 27, 10]. On one hand, ROLAP scales better with the amount of data and is better at handling models with very high cardinality dimensions when compared with MOLAP[5]. On the other hand, MOLAP is regarded to have faster "query response times" [22]. Hybrid OLAP (HOLAP) which "allows storing part of the data in a MOLAP store and another part in ROLAP store" provide both scalability on size of data and performance advantage on specified part of data in MOLAP area.

Traditionally, an OLAP application works with an independent database and

uses an extract-transform-load (ETL) tool to import data from OLTP applications. The requirement of real-time OLAP application is gradually drawing more and more attention and thus leading many researchers and companies to devote resources toward the implementation of OLAP and OLTP integrated systems [31]. Santos *et al.* propose an integrated OLTP and OLAP framework to respond to OLAP queries in real-time [26]. To solve the dilemma mentioned in Section 2.2, they use a temporary table to hold records inserted after the last integration phase to avoid expensive insertions in the original tables due to the indexes for OLAP queries on it. Their approach merges the temporary table with the original table when the temporary table is too large to yield a performance benefit.

2.4.2 Approximate Computing

An important issue in the development of real-time OLAP systems is the synchronization between changes caused to the database system by the processing of transactions and the calculation of cell values to respond to user queries. Ungar *et al.* propose a synchronization-free invalidation policy that does not synchronize transaction processing with query processing [30]. This policy leads to inconsistent cell values in the database. They introduce the concept of *fresheners* to reduce the level of inconsistency and to provide more accurate answers to the queries. A freshener is an asynchronous thread that recalculate the value of a stale computed cell. Several scheduling policies can be used for freshener threads. They built a simple prototype and evaluated the performance and level of inconsistency on a simple data set. Their limited experimental evaluation pointed to promising scalability improvement and acceptable levels of inconsistency. This thesis is inspired by Ungar's work and the research reported here was performed in collaboration with the same IBM team. We developed a new prototype to investigate the levels of inconsistency and the performance effects of the synchronization-free invalidation policy. Compared to Ungar's prototype, our prototype is implemented with a lower-level program language (Ungar *et al.* use smalltalk and we use C++) that gives us more control over the data structures. Additionally, our prototype is able to load the dataset along with rule files dumped by TM1 to enable us to experiment with a dataset from the daily operation of a very large corporation. Finally, we propose to use freshener to collect inconsistency level while the program is running and to adjust the number of fresheners according to the collected information.

Kiviniemi *et al.* propose a real-time OLAP that tolerates inconsistency and attempts to reduce recalculation [18]. Their approach skips re-computation when the difference between the result after update does not exceed an user-specified toleration range. This mechanism reduces the overall recalculation effort and thus improves throughput. In contrast, the approach investigated in this thesis removes part of the synchronization overhead to improve scalability, which is not a central concern for their mechanism. Kiviniemi *et al.* assume that an update will not cause significant difference to a cell A if it does not cause significant difference for A's child cell B. Therefore, their system reduces overhead by skipping the computation of the difference of all cells whose child cell is not significantly affected by the update. Such assumption only holds when computed cells may only be calculated with a very limited set of functions(*e.g.* sum, average). In our prototype where user are allowed to define rules with arbitrary functions, their assumption is not valid.

Chen *et al.* implements a near-real-time data warehousing system [11]. They define metrics to measure the affect of update on pre-computed results and then decide if a fresh computation is needed. Similar to Kiviniemi's lazy-aggregation approach, this implementation reduces the computation needed but does not affect scalability.

Relaxing synchronization requirements is not an exclusive domain of database-related applications. Rather, it is also a major concern for general-purpose parallel programming and automatic parallelization of sequential programs and it becomes more important with more processers you have. Renganarayana *et al.* and Misailovic *et al.* present frameworks to relax the synchronization in general programs for better scalability [25, 21]. The system developed by Renganarayana *et al.* allows users to select a profitable degree of relaxation to determine the frequency of synchronization avoidance. We may apply a similar idea in the future to select a degree of synchronization relaxation in our approach to control the inconsistency level where the lower degree of synchronization results in higher level of throughput and chance to get incorrect results. Misailovic *et al.* define an accuracy metric and use statistic analysis to understand the effect of synchronization relaxation. Later they introduce QuickStep, a system to parallelize sequential programs [20]. Different from standard parallelizing compilers, QuickStep tolerates certain level of inconsistency, in the sense that the output could be different from the output produced by a sequential program to some acceptable level, and thus generate code

with more parallelism. The study in this thesis is focused specifically on eliminating synchronization on the invalidation procedure of OLAP application with cache system, and thus has more opportunity for specialized improvements that cannot be easily applied to the relaxation of synchronization on a general program.

The metric used by Misailovic *et al.* to measure accuracy is based on the value of the output [21, 20]. Such metric may be the most reasonable choice for a general program but is not the best for an OLAP application. This thesis defines a probability-based metric to measure the inconsistency level.

2.4.3 Incremental Computation

Incremental computation is a research area that deals with the problem of how to update the result of a computation in an efficient manner in the face of changes to the input of that computation [24]. Implementations of incremental computation rely on function caching to obtain incremental evaluation [24, 6, 16] . Acar *et al.* introduce dynamic dependency graphs (DDGs) and combine DDGs with a memorization technique to create what call *self-adjusting computation* [8, 7]. Recently, Hammer proposed a composable, demand-driven, incremental computation approach. Their prototype, named ADAPTON, further improves efficiency by taking advantage of lazy evaluation in the propagation algorithm [15].

Unfortunately, it is non-trivial to apply existing incremental computation techniques to a real-time OLAP cache system because of the constrained memory space and the large volume of data processed by real-time OLAPs used by large corporations. Incremental computation and self-adjusting computation systems rely on some form of dependency graph to record the relation between nodes for re-computation. In an OLAP application, each cell must be regarded as a node in such graph because each cell may affect the value of other cells. Often the number of cells is so large that a dependency graph, either static or dynamic, is too large to fit in memory. It is possible to group the cells into areas, to record the dependency graph of those areas, and then to apply an incremental-computation technique to this area graph to reduce the time needed for individual re-computations. But such an approach would not produce the scalability that synchronization elimination attempts to achieve and therefore is not considered in this thesis.

Chapter 3

Synchronized Policies to Maintain Consistency of Cached Cells

Real-time OLAP applications use a type of memorization. They rely on a cache system to store the results of previously answered queries on computed cells to accelerate future queries and to improve overall throughput. The value cached for a computed cell is stale if it depends — either directly or indirectly, via other computed cells — on the values of entered cells that have been updated since the last time that the cached value was computed. To maintain the correctness of cached values, stale cached cells must be either invalidated or re-calculated, re-calculating the value based on recent updates is also called *refreshing*. This chapter reviews several cached-value maintenance policies. Then the performance of an OLAP application using each of these policies will be compared with an OLAP application that does not use a cache system in Chapter 6.

3.1 Motivating the Need for Synchronization

This section motivates the need for synchronization in OLAP by presenting an example where invalidating the value of cached cells without synchronizing leads to stale values in computed cells that appear to be updated. First, we present a synchronization-free invalidation policy. In this policy, whenever an update — insertion, deletion or value change — occurs on an entered cell A, the application enumerates all the positions of the cube that could contain computed cell that depend on the value of A and invalidates those cached cells.

Algorithms 1 and 2 show the procedure to update the values of entered cells and

Table 3.1: Illustration of Thread Safety Violation

Thread 0	Thread 1
A.cached_value ← calculated()	
	B.value ← new_value
	A.is_valid ← false
A.is_valid ← true	

computed cells. The calculation of the new cached value in line 1 of Algorithm 2 uses the most up-to-date values. These algorithms are not thread safe because they could allow a stale value in a cached cell that appears to be valid. Consider the scenario illustrated in Table 3.1 where computed cell A depends on entered cell B. Assume that thread 0 is processing a query on A at the same time that thread 1 is updating the value on B. Assume that initially thread 0 is at line 1 in Algorithm 2 and has already retrieved the value of B. Then thread 1 finishes updating the value of B and executes the for loop in algorithm 1 thus changing the value of the `is_valid` flag to `false` before thread 0 finishes the calculation. After that, thread 0 finishes the calculation and changes the `is_valid` flag to `true`. Now the cached value of A is stale because it is not using the most up-to-date value of B that thread 1 just updated. However the value of A is marked to be valid.

Algorithm 1 Entered_Cell::update(double new_value) Sync-Free Invalidation Policy

```
1: value ← new_value;
2: cell_array cells_to_invalidate ← enumerate_cells();
3: for i in cells_to_invalidate do
4:     i.is_valid ← false;
5: end for
```

Algorithm 2 Computed_Cell::cache_value()

```
1: cached_value ← calculate();
2: is_valid ← true;
```

Enumerating all the cached cells that depend on a given entered cell, as required by the statement in line 2 of Algorithm 1, is not trivial and results in significant overhead. The combination of the coordinates and its direct and indirect parents of all dimensions may result in a very large set of coordinate tuples. Thus, it is time consuming to iterate over this set for all cached cells and holding a lock during this long time procedure is extremely harmful to the scalability. Specially designed data

Algorithm 3 Entered_Cell::update(double new_value) Synchronized Invalidation Policy with Global RW-Lock

```
1: global_rwlock.acquire_wrlock();
2: value ← new_value;
3: cell_array cells_to_invalidate ← enumerate_cells();
4: for i in cells_to_invalidate do
5:     i.is_valid ← false;
6: end for
7: global_rwlock.release_wrlock();
```

Algorithm 4 Computed_Cell::cache_value() Synchronized with Global RW-Lock

```
1: global_rwlock.acquire_rdlock();
2: cached_value ← calculate();
3: is_valid ← true;
4: global_rwlock.release_rdlock();
```

Algorithm 5 Entered_Cell::update(double new_value) Synchronized Invalidation Policy with Per-Cell Lock

```
1: value ← new_value;
2: cell_array cells_to_invalidate ← enumerate_cells();
3: for  i in cells_to_invalidate do
4:     i.acquire_percell_lock();
5:     i.is_valid ← false;
6:     i.release_percell_lock();
7: end for
```

Algorithm 6 Computed_Cell::cache_value() Per-cell Lock Policy

```
1: acquire_percell_lock();
2: cached_value ← calculate();
3: release_percell_lock();
```

Table 3.2: Illustration of Thread Safety Violation with Per-cell Lock

Thread 0	Thread 1
A.value ← new_value	
C.is_valid ← false	
	C.cached_value ← calculate();
B.is_valid ← false	C.is_valid ← true

structures and optimizations may reduce the overhead on this procedure, but such optimizations were not explored in this thesis.

3.2 Synchronized Invalidation

A global read-write lock can be used to prevent the update of the value in an entered cell and the calculation of the value in a dependent computed cell from happening simultaneously. The policy, named Synchronized Invalidation(SI), described in this section allows multiple calculations, or a single update, to execute simultaneously. This policy is suitable for application workloads with more queries than updates.

In Algorithm 3 for update, the thread acquires a write lock on the global read-write lock at line 1 and releases that lock at line 7. In line 3, all the cells that are dependent on `this` entered cells are enumerated and added to the array of cells to be invalidated. In Algorithm 4 for cached value, the thread acquires the read lock at line 1 and releases it at line 4. The calculation of the new value to be caches in line 2 uses the most up-to-date data.

Managing the global lock is the main bottleneck for scalability. An alternative is to use a separate lock for each cell (Algorithms 5 and 6) to allow multiple non-conflicting computations and updates to happen at the same time without resulting in incorrect stale cells. But separate locks could not prevent the staleness caused by indirect dependency — dependencies through other computed cells instead of direct dependency on entered cells.

Suppose we have entered cell A and computed cells B and C. B depends on A, C depends on B and thus depends on A indirectly. Assume that the value of B is already calculated and cached. Then, for the scenario demonstrated in Table 3.2, thread 0 updates A and thus invalidates C and B. Assume that thread 0 changes the value and invalidates C first. Then thread 1 calculates the value of C. It uses B's old cached value because it appears to be valid, but B's cached value is actually stale because of the update on A. Thread 1 then finishes the calculation and caches a stale result but records it as fresh. Thread 0 then invalidates B. As the result, cell C is calculated using out-of-date value of cell B and appears to be valid.

Algorithm 7 solves the incorrect bookkeeping of staleness illustrated above by locking all the dependent cells first at line 3 to line 5 and then invalidating them one by one. But this policy leads to the dead-lock scenario described in Table 3.3.

Algorithm 7 Entered_Cell::update(double new_value) Alternative Per-cell Lock Policy

```
1: value ← new_value;
2: cell_array cells_to_invalidate ← enumerate_cells();
3: for i in cells_to_invalidate do
4:     i.acquire_percell_lock();
5: end for
6: for i in cells_to_invalidate do
7:     i.is_valid ← false;
8: end for
9: for i in cells_to_invalidate do
10:    i.release_percell_lock();
11: end for
```

Algorithm 8 Entered_Cell::update(double new_value) Synchronized Cache Clearing Policy

```
1: global_rwlock.acquire_wrlock();
2: value ← new_value;
3: delete_all_cached_cells();
4: global_rwlock.release_wrlock();
```

Algorithm 9 Computed_Cell::cache_value() Synchronized Cache Clearing Policy

```
1: global_rwlock.acquire_rdlock();
2: cached_value ← calculate();
3: global_rwlock.release_rdlock();
```

Table 3.3: Illustration of Die Lock with Per-cell Lock

Thread 0	Thread 1
A.value ← new_value	
B.acquire_percell_lock();	
	C.acquire_percell_lock();
C.acquire_percell_lock()	B.acquire_percell_lock();
	inside C.cached_value ← calculate();
Dead Lock!	

3.3 Clear All the Cached Values

An alternative approach, to avoid the complexity of enumerating all dependent cached cells for a given entered cell, is to clear the cached cell set, *i.e.* to delete all cached values when any update happens. This policy still requires a global read-write lock that needs to be acquired while the deletion of all cached values takes place (line 1 and 4 in Algorithm 8). During the computation of a cached value a read lock must be acquired (line 1 and 3 in Algorithm 9) to avoid using the cached cells that are deleted.

This chapter introduced several invalidation policies used during update to maintain the correctness of cached values. It also discussed the necessity for a synchronization mechanism to prevent the use of stale values in the computation of query responses. Unfortunately these mechanisms, such as the global read-write lock, lead to significant overhead and thus reduce scalability dramatically. The next chapter introduces inconsistency-tolerating approaches that eliminate the synchronization mechanism to achieve better scalability.

Chapter 4

Inconsistency-Tolerating Policies

Inconsistency-tolerating policies are introduced to reduce synchronization overhead and improve the throughput in OLAP systems at the cost of introducing some level of inconsistency. This chapter examines two classes of inconsistency-tolerating policies: a synchronization-free policy and an invalidation-free policy. The idea is to allow the use of cached values to compute the query results without requiring that these values be synchronized with the most recent data entered into the OLAP system. The drawback of this approach is that many queries could return stale results. A proposed solution to reduce the occurrence of stale query results, while keeping the synchronization cost low, is to introduce specialized threads that periodically recompute the cached value for cells. These specialized threads are called *Fresheners*. There are several parameterized policies that can be used to determine how many fresheners should be used in a system and what is their approach to select cells to be recomputed. This chapter introduces inconsistency-tolerating invalidation policies and fresheners, discusses freshening policies, and introduces metrics that are useful for the measurement of the level of inconsistency in the system.

4.1 UI Policy

Sections 3.2 and 3.3 established that synchronization during invalidation generates heavy overhead in a system operating with multiple threads. Ungar *et al.* propose an approach to tolerate inconsistency that aims to achieve scalability on massively-parallel low-latency systems[30]. This approach uses an Unsynchronized Invalidation(UI) policy. This policy performs invalidation without synchronization,

as described in Section 3.1, while updating a cell. Then the policy relies on *fresheners* to reduce the number of cells that are stale and the number of queries that return an incorrect result.

4.1.1 Freshener

While motivating the need for locks, Section 3.1 described a scenario where an invalidation procedure fails to invalidate a cached cell because of a lack of synchronization. As the result, a valid cached cells that is used to accelerate future queries may contain stale value. To address this problem, we use the idea of Ungar *et al.* and introduce freshening threads to recompute the value of cached cells that are marked valid in case they in fact hold stale values and then freshen stale ones as shown in Algorithm 10.

Algorithm 10 Cell::fresh()

```
1: if is_valid() then
2:     tmp ← recompute();
3:     if cached_value ≠ tmp then
4:         cached_value ← tmp;
5:         stale_num ← stale_num + 1;
6:     end if
7:     total_freshned ← total_freshned + 1;
8: end if
```

Ungar *et al.* describes two policies that can be used by a freshener to select the order in which cached cells are recalculated: round-robin and random [30]. In the round-robin policy, the freshener thread visits the cells in the cached-cell set one by one and recalculates the ones that have the valid flag. In the random policy, each time the freshener selects a valid cell randomly from the set of cached cells, and recalculates value. The prototype that we built for this thesis uses the round-robin policy along with UI policy, which appeared to be efficient enough to fix the staleness issue introduced by lack of synchronization.

4.2 NINS Policy

The previous section introduced the UI policy proposed by Ungar *et al.* Their policy dramatically increases the scalability of OLAP systems when compared to the synchronized policies discussed in Chapter 3. But even without synchronization, it is still time consuming to locate the computed cell to be invalidated. To further reduce

the overhead, we propose the No Invalidation and No Synchronization(NINS) policy that entirely eliminates the invalidation procedure, to avoid its overhead, when an entered-cell value is updated.

Unfortunately, without invalidation a very large number of valid cached cells would have to be stored in the system. Therefore, fresheners would not be able to visit all valid cells, within a reasonable time frame, using a round-robin policy to re-calculate the values of valid cells. Moreover, it may take more time for a freshener to re-compute the value of a cell than it would take a worker to compute the value of the same cell because a freshener does not use any cached value to ensure the result's correctness.

4.3 Most-Recently-Queried (MRQ) Freshening Policy

Therefore, we propose a Most-Recent-Queried (MRQ) policies to be used by fresheners. The idea is that only the cached cells that have been queried by the K most recent queries should be recalculated. Cells that were not queried recently may either remain valid but not be freshened, or else they may be invalidated, depending on the specific variation in the refreshing policy. With different definition of 'recent' access cells, we proposed an Access-Counter-Based MRQ policy and the Time-Based MRQ policy in the rest of this section.

With the MRQ freshening policy, fresheners focus on recomputing the value of recently queried cells hence more frequently accessed cached cells have a higher chance of being re-computed. A cached cell that has not been accessed for a long time has higher probability of becoming stale, and thus of being invalidated by the freshener, to both reduce the possibility of generating stale results and reduce the size of the set of valid cells that fresheners have to visit.

4.3.1 Access-Counter-Based MRQ

In Access-Counter-Based MRQ policy, a global counter, *global_access_counter*, records the total number of access on all cached cells. And the local counter, *local_mr_access*, is a per cell counter that for a cached cell CC, *local_mr_access* records the number of access on all the cached cells when CC is accessed last time. Whenever a worker uses the cached value of cell, it increase the *global_access_counter* by one. This incrementation and fetching requires an atomic operation, fetch_and_add, and can be a concern for scalability in large-scale shared-memory systems. The query thread

then stores the result in cached cell's *local_mr_access* after divide it by K. To reduce the cost of this computation the value of K is an integer power of two, thus allowing the division to be performed as a logical shift-right operation(Algorithm 11).

Algorithm 11 Cell::get_cached_value()

```
1: local_mr_access ← fetch_and_add(global_access_counter ,1) ≫ log(K)
2: return cached_value
```

A freshener thread repeatedly randomly selects a cell in the cached-cell sets, and checks the cells *local_mr_access*. There are three possible actions for the freshener: (1) invalidate the cell; (2) do nothing; and (3) recalculate the value of the cell. It only freshens a cell if its *local_mr_access* equal to the *global_access_counter* divided by K (line 1 in Algorithm 12). For the sake of efficiency, K must be an integer power of two and thus the division can be implemented as a logic right shift of log(K). A cached cell is invalidated if the difference between *local_mr_access* and *global_access_counter* divided by K is larger than M/K, where M is an algorithm parameter that is a multiple of K (line 2-4 in Algorithm 12). In the current implementation of the prototype the value of M/K is set to 16.

Algorithm 12 Cell::fresh()

```
1: if local_mr_access <> global_access_counter/K then
2:     if global_access_counter/K − local_mr_access > M/K then
3:         invalidate();
4:     end if
5:     Return
6: end if
7: if is_valid() then
8:     tmp ← recompute();
9:     if cached_value ≠ tmp then
10:        cached_value ← tmp;
11:        stale_num ← stale_num + 1;
12:    end if
13:    total_freshned ← total_freshned + 1;
14: end if
```

4.3.2 Time-Based MRQ

This approach uses a time interval T instead of K and still keep the *global_access_counter* and *local_mr_access*. But the *global_access_counter* is no longer increased by the worker when a reference to a cached cell happens. Instead it automatically increases

by one for every T time. In this approach, only the Master thread is responsible to update the *global_access_counter* periodically and workers and freshener threads only read from the *global_access_counter* . The goal is to improve scalability, in comparison with the access-counter based approach described in Section 4.3.1, on large-scale shared-memory machine. The local copy of *global_access_counter* in each worker remains coherent, and thus can be read multiple times without any need for synchronization or communication, as long as the value of the *global_access_counter* does not change. Whenever the value of the *global_access_counter* is changed by the master thread, this write operation will invalidate all the local copies of the *global_access_counter*. Therefore, on the next read of *global_access_counter* each worker will fetch the new value.

Algorithm 13 details how the master thread increments the *global_access_counter* every T time. In this approach, during the freshening, the freshener thread checks if a cached cell was accessed within the last T time by comparing its *local_mr_access* and *global_access_counter*, as shown Algorithm 14, and then decides whether the cell should be freshened, invalidated or left alone.

Algorithm 13 Master::update_gac()

```
1: while TRUE do
2:     sleep(T)
3:     global_access_counter++
4: end while
```

Algorithm 14 Cell::fresh()

```
1: if local_mr_access <> global_access_counter then
2:     if global_access_counter/K − local_mr_access > M then
3:         invalidate();
4:     end if
5:     RETURN
6: end if
7: if is_valid() then
8:     tmp ← recompute();
9:     if cached_value ≠ tmp then
10:        cached_value ← tmp;
11:        stale_num ← stale_num + 1;
12:    end if
13:    total_freshned ← total_freshned + 1;
14: end if
```

Both Access-Counter-Based MRQ and Time-Based MRQ require each cached

cell to store the *local_mr_access* that generate extra space requirement. In our current experiments, such space overhead had no significant impact on memory consumption.

The MRQ is a simple heuristic for the selection of cached cells to re-calculate. It assumes that the most-recently queried cached cells are more likely to be used in the future. Therefore, the MRQ policy is only reasonable for workloads where such assumption holds most of the time. The investigation of alternative policies for systems where this assumption does not hold is left for future work.

4.4 Measuring Inconsistency

Inconsistency-tolerating policies result in stale cached values. Therefore, from a user's perspective it is crucial to measure the level of inconsistency in a system that allows stale values to be used to satisfy queries. Intuitively, a user wants to know how often does the system answer a query based on stale values? To answer this question, we define the metrics *Query Stale Rate* (QSR) and *Cached-Value Stale Rate* (CVSR).

4.4.1 Query Stale Rate

The QSR metric indicates the proportion of queries that returned a stale value. QSR is computed as the following ratio:

$$\text{QSR} = \frac{\text{Number of stale query results}}{\text{Total number of queries}} \tag{4.1}$$

Unfortunately, it is non-trivial to detect the staleness of results accurately without a time-stamp based log system. In order to obtain a good estimate for QSR, our prototype implements an auto-recomputing mode to allow workers to automatically recompute the value of a query without using any cached values after answering each query. The drawback of this implementation is that the computation of the value to verify if it was stale changes the throughput of query processing and thus alters the operation of the system. It also changes the order between queries and updates because the queries take longer to be processed. In all experiments in Chapter 6, the prototype only uses auto-recomputing mode when measuring the query stale rate.

Given that more updates may arrive while the same number of queries is processed because of the lower query processing throughput, the QSR measured using

this method is likely to be higher than the actual QSR for the normal operation of the system.

4.4.2 Cached-Values Stale Rate

The CVSR metric indicates the proportion of cached cells that have a stale value. It is computed as the following ratio:

$$\text{CVSR} = \frac{\text{Number of stale cached cells}}{\text{Number of valid cached cells}} \tag{4.2}$$

Theoretically, the CVSR is computed for a given snapshot of the system. Its value can be used to compute the probability of getting a stale result if a cached-cell value is used. Given that a query on the value of a single computed cell could use the value of k cached cells, the probability that an answer to such a query is stale is $1 - (1 - \text{CVSR})^k$ because the answer will be stale if any of the cached values that it uses is stale.

Stopping the entire OLAP system to calculate the CVSR is not realistic. Therefore, we propose a sampled CVSR where fresheners collect the information to compute CVSR. This sampling technique has a much lighter overhead. In Algorithm 14, the freshener records the number of stale cached cells and the total number of cached cells that are freshened (lines 11 and 13). The sampled CVSR is defined as the ratio between these numbers according to the following equation:

$$\text{sampled CVSR} = \frac{\text{Number of stale freshened cached cells}}{\text{Number of freshened cached cells}} \tag{4.3}$$

Herein CVSR refers to sampled CVSR because it is the only one available while the system is running.

4.5 Summary

This chapter introduces two inconsistency-tolerating policies along with metrics to measure the level of inconsistency in the system. The fresheners play a key role in fixing stale cached values. The chapter then discussed two MRQ freshening policies to improve the freshener efficiency.

Chapter 5

Adaptive Inconsistency Level Framework

The NINS policy in Chapter 4 delivers throughput improvement at the cost of incorrect query results. The number of incorrect results generated has a close relation with the workload processed by the OLAP system. This chapter presents the Adaptive Inconsistency Level Framework (AILF) that aims to maintain the inconsistency to an acceptable level, specified by the user, under various workload. It collects information about the inconsistency level in the last period and adjusts the number of fresheners to affect the inconsistency level in the next period.

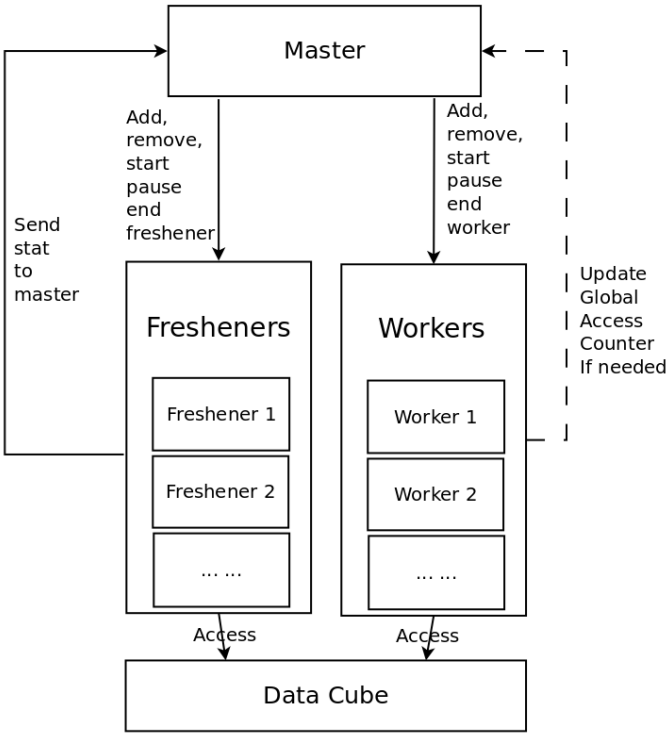
5.1 Framework Description

As in Figure 5.1, the AILF consists of three parts: a Master, a groups of Workers and a group of Fresheners. Each individual thread is executed by either a worker or a freshener. Each worker is responsible to answer user queries and execute update. Fresheners keep re-computing the cached values to update stale values among them. The Master monitors CVSR and adjusts the number of fresheners to affect the inconsistency level.

5.1.1 Worker

A worker answers user's query and executes update. The worker executes an infinity loop where it listens to commands from the Master by checking its working phase parameter (line 2 in Algorithm 15). The worker executes jobs whenever Master sets its state to *Working*. When a worker receives a *Pause* command from the Master, it changes its working phase to *Paused* to let the Master know that has paused. Then

Figure 5.1: AILF structure



Algorithm 15 Worker::run()

```
1:  while True do
2:      switch *my_working_phase do
3:          case Working
4:              execute_next_job();              ▷ Answer Query or execute update.
5:          case Pause
6:              *my_working_phase ← Paused;
7:              continue;
8:          case Paused
9:              continue;
10:         case End
11:             *my_working_phase ← Ended;
12:             return;
13: end while
```

the worker continues to busy waiting until a different command is received. The worker terminates when it receives an *End* command.

5.1.2 Freshener

Algorithm 16 Freshener::run()

```
1: i ← 0;
2: cached_cell_list ← get_all_cached_list();
3: while True do
4:     switch *my_working_phase do
5:         case Working
6:             if i > cached_cell_list.size() then ▷ Get new list of cached values to refresh if current list are done.
7:                 free(cached_cell_list);
8:                 cached_cell_list ← get_all_cached_list();
9:                 i ← 0;
10:            end if
11:            cached_cell_list[i].fresh();
12:            i++;
13:        case Pause
14:            *my_working_phase ← Paused;
15:            continue;
16:        case Paused
17:            continue;
18:        case End
19:            *my_working_phase ← Ended;
20:            return;
21: end while
```

As described in Algorithm 16, similar to a worker, the freshener receives and reacts to commands from the Master in every iteration of the infinity while loop. Compared to the worker, the freshener would select the next value from the current cached cells set to re-compute and get a new set of cached cells when it gets to the end of current list. The number of cached values been freshened and the number of stale ones among them are recorded to report current inconsistency level.

5.1.3 Master

The Master is able to add or remove, start, pause or end individual worker or freshener threads. It can also get statistic information about performance from workers or inconsistency information (CVSR) from fresheners.

The Master controls workers and fresheners through the content of shared arrays

thread_idx_status and *thread_work_phase*. Both arrays are initialized to be all zeroes. When creating either a worker or a freshener, the Master looks for the first element i in array *thread_idx_status* which is zero (denote that the index is available) and set it to special value (e.g. 1 for worker, 2 for freshener). The worker or freshener will keep the address of its thread_idx_status element (*&thread_idx_status[i]*) for future usage. As described in Algorithms 15 and 16, once it is started, a worker or freshener runs an infinite loop and executes jobs according to the state set in *thread_work_phase[i]*, a local copy of which is maintained in *my_working_phase*. The Master controls existing workers and fresheners by changing the content of array *thread_work_phase*. It could change *thread_work_phase[i]* to *Pause* which let the corresponding worker or freshener keep busy waiting after finish current on-going job and change *thread_work_phase[i]* to paused to notify the Master. It could also change it back to working thus it would continue to perform querying and updating or recomputing. Ending makes a worker or freshener exit after it finishes the current job. The Master is responsible for deleting the ended worker or client instance and to set the corresponding element in *thread_idx_status* and *thread_work_phase* back to zero for future usage. The Master can only create or delete worker and freshener sequentially because no synchronization techniques are applied on *thread_idx_status* and *thread_work_phase*.

Algorithm 17 Master::add_a_worker

```
1: for i ← 1 to MAX_THREAD_NUM do
2:     if thread_idx_status[i] = 0 then
3:         new_worker ← new Worker(i);    ▷ New instance of worker with thread index i
4:         workers_list.push_back(new_worker);
5:         Return;
6:     end if
7: end for
```

5.2 Adaptive to various workload

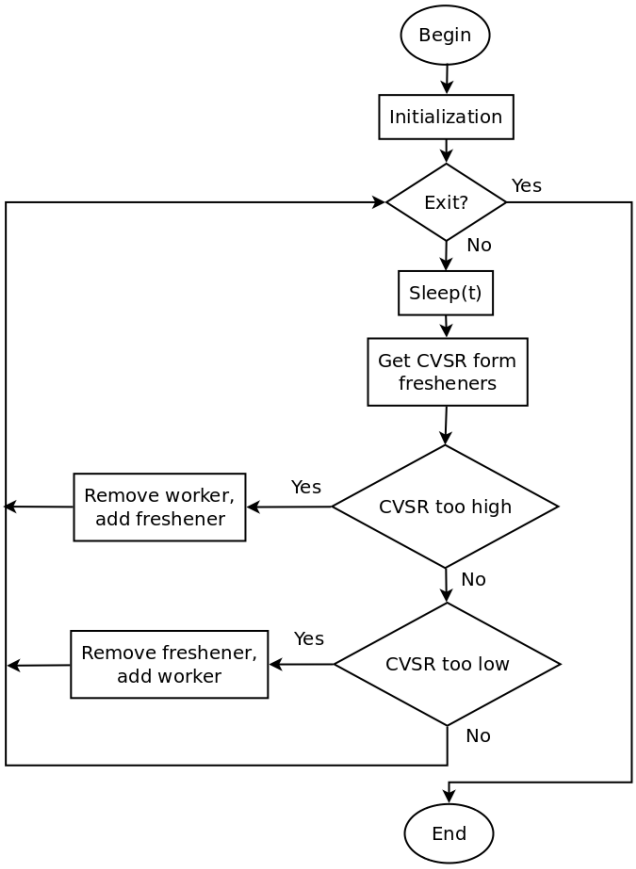
As its name implies, the AILF can adapt to various workloads with a predefined level of inconsistency by adding or removing fresheners and workers (Figure 5.2).

The user sets a target CVSR T and tolerance range ϵ. Then the AILF system tries to keep the average CVSR between $T \pm \epsilon$. The Master in the AILF system initially creates several workers and fresheners. Then it let all the workers and

Algorithm 18 Master::remove_a_worker

1: $worker\_to\_delete \leftarrow workers\_list.pop()$;
2: $i \leftarrow worker\_to\_delete.get\_thread\_idx()$;
3: $thread\_work\_phase[i] \leftarrow End$;
4: **while** $thread\_work\_phase[i] != Ended$; **do**
5: ; ▷ Busy wait until the worker finished current job and Ended
6: **end while**
7: $thread\_idx\_status[i] \leftarrow 0$;
8: $thread\_work\_phase[i] \leftarrow Nul$;

Figure 5.2: AILF Work Flow



fresheners start their infinity loop in their *run()* function. After that, the Master sets all workers and fresheners' state to *Working* so that the system can answer queries and execute updates required by user. The Master runs in a loop: 1) sleeps for pre-defined time interval, 2) gets statistic result of CVSR from fresheners for the interval during which Master was sleeping, 3) replaces a worker with a freshener if $CVSR > T + \epsilon$, 4) replaces a freshener with a worker if $CVSR < T - \epsilon$. The infinity loop may terminate at the beginning of every iteration.

Currently, AILF, as a self-regulating system, only allows the master thread to add or remove a single freshener on each iteration. Future extensions of this prototype may allow more sophisticated controlling strategy for AILF.

The user of this system cares more about QSR than CVSR but we still choose to control the CVSR for following two reasons. Firstly, it is hard to measure QSR on running system because there is no way to detect a stale query result without extra overhead. Other than that the QSR is related with CVSR and stable CVSR helps to maintain a stable QSR. On the other hand, it is also possible to implement an on-line sample QSR system that randomly selects some queries and rechecks them without using any cached values to obtain a sample QSR. On such system, the program may use the sample QSR instead of CVSR to control the inconsistency level.

Chapter 6

Evaluation

This chapter presents a performance evaluation of the policies studied or introduced in this thesis. When studying these policies, an important issue to investigate is how much inconsistency is actually introduced by the inconsistency-tolerating policies. It is also important to know wether the elimination of synchronization overhead delivers scalable performance in OLAP systems while keeping inconsistency to acceptable levels. The effectiveness of the adaptive framework presented in Chapter 5 to control the inconsistency level in an OLAP system is also of interest to researchers and practitioners. This performance evaluation study confirms some of the expected performance trends for the policies, and yields two important findings. The main results of this evaluation can be summarized as follows:

- **Confirming expected performance trends:**
  - Synchronized policies scale poorly and fail to provide acceptable performance under random access pattern. In contrast, inconsistency-tolerating policies dramatically improve the scalability.
  - The policy that performs no invalidation and no synchronization further reduces the synchronization overhead in comparison to the policy that performs unsynchronized invalidations, but the elimination of invalidation does introduce higher levels of inconsistency.
- **Adding more fresheners reduces the inconsistency level.** This finding is true for the policy that does no invalidation and no synchronization (NINS policy). Unfortunately, the NINS policy's ability to affect the inconsistency level through management of the number of fresheners is limited. Thus the NINS policy does not achieve the same level of inconsistency as the policy that

does unsynchronized invalidation (UI policy). But the NINS policy produces better throughput, even when using fewer workers, than the UI policy.

- **Inconsistency Level Stability:** While the Adaptive Inconsistency Level Framework (AILF) needs some time to collect information about the inconsistency level — and thus cannot adapt at the start of the operation of the system, after it has collected enough feedback it is able to maintain the inconsistency level stable.

The remainder of this chapter describes the experiments and presents the experimental results that support these observations. First the experimental evaluation platform is described in Section 6.1. Section 6.2 describes the update policies and Section 6.3 presents the data sets used for the experimental evaluation. After that the chapter presents the experimental results that support the findings listed above.

6.1 Experimental evaluation platform

This experimental evaluation uses an IBM 8233-E8B system that has 32 3.55 GHz POWER 750 processors with SMT disabled and 512 GB memory running AIX 7.1.0.0. The program is compiled by xlC with optimization level O5 and `-qarch=pwr7 -qtune=pwr7` options to allow the compiler to optimize the prototype for the POWER 7 architecture.

6.2 Update Policies

All the update policies that appear in this evaluation are listed below. The short names in parenthesis are consistently used as legend in all the graphs.

- `No Cache (NC)`. The system does not record any results of queries and thus does not need any invalidation or recalculation during update.
- `Synchronized Cache Clearing (SCC)`. When an update occurs on an entered cell, clear the whole set of cached cells. All threads are synchronized with a read-write lock — described in Section 3.3.
- `Synchronized Invalidation (SI)`. When an update occurs on an entered cell, invalidates only the cache entries for the computed cells whose value is

affected by the updated entered cell. All threads are synchronized with a read-write lock — described in Section 3.2.

- `Unsynchronized Invalidation (UI)`. In this synchronization-free policy, when an update occurs on an entered cell, invalidate the cached computed cells whose value is affected by the updated entered cell. Threads are NOT synchronized and the system relies on fresheners to ammeliorate staleness — described in Section 4.1.

- `No Invalidation and No Synchronization (NINS)`. In this synchronization-free policy, when an update on an entered cell occurs, do not invalidate or re-calculate any cached cell and completely rely on fresheners to reduce staleness — described in Section 4.2. Three different legends appear in the graphs for this policy because there are three different freshening policies (Round-Robin, Access-Counter-Based MRQ and Time-Based MRQ):

  - NINS - R denotes the use of the round-robin freshening policy;
  - NINS - C with K=256 denotes the use of the Access-Counter-Based MRQ policy with K equal to 256; and
  - NINS - T with ti=0.5 denotes the use of the Time-Based MRQ policy with a time interval equal to 0.5s.

In the current prototype there is no attempt to optimize these policies beyond the description provided in this document. There is room to improve these policies through optimizations, but this is left for future work.

6.3 Data Sets

Two different data sets are used for the experimental evaluation. The first one is a simple data set that is useful to study properties of the policies. The second data set contains actual data collected from the operation of a major manufacturing company. Due to confidentiality issues we cannot disclose the name of the company that provided the data.

- *Simple Data Set*

  This data set contains two data cubes that are part of an IBM Cognos TM1 demo data set named `s_data`. The first data cube contains stable data that

does not change throughout the experiments. This cube has four dimensions, and each dimension may have from three to fifty one coordinates. The average depth (average of each leaf nodes' height in the dimension tree) of these four dimensions is 3.5. This cube contains 8355 entered cells that feed values to 43,964 computed cells.

The second data cube is where the program performs update and queries in the experimental evaluation presented in this chapter. This cube contains one additional dimension compared to the previous data cube which has six coordinates. This cube has 16,786 entered cells that feed 298,568 computed cells. This cube also uses data from the first cube described above for calculation.

- *Manufacturer Data Set*

  This data set contains two data cubes that are part of a manufacturing company's product information management system. The experiments reported in this thesis use one tenth of the entered cells in the actual data set. Currently, the prototype is not able to load all data cubes that we received from the company in a reasonable amount of time because it needs to pre-process the data to accelerate the calculation of cell values. The prototype requires more than twenty minutes to load the two cubes that we use and would require much more than two hundred minutes to load all the cubes that we received — the time required to load the cubes grows super-linearly with the number of entered cells. Moreover, in the current prototype setup the entire data set must be loaded in order to run each experiment.

  In this data set, one data cube contains the stable data with no change in all the following experiments. It has two dimensions one with two coordinates and another with 51 coordinates with depth equal to six. It contains only ten entered cells and one computed cell.

  Another data cube is where the program performs updates and queries. It includes five dimensions that contain from 2 to 2719 coordinates in each of them (2, 420, 478, 1190 and 2719 coordinates). The average depth of them is 4.1. This cube has 26598 entered cells that related with 9287049 computed cells.

For the experiments that use these data sets, we use synthetic workload where we can control the update ratio and the locality of reference of access in the data cubes. It would be desirable if future research could use workload that is representative of actual production usage of these data sets. However, such workload was not available for our evaluation.

6.4 Experiments Confirming Expected Trends

This section reports the results of two experiments that study the scalability of the various policies and the impact of inconsistency-tolerating policies on throughput and inconsistency level in the system. These results confirm expectations in relation to the performance of the various policies.

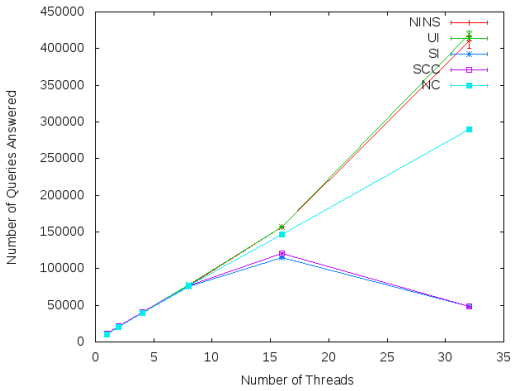
6.4.1 Scalability Study

The results of the experiment described in this section confirm that synchronized policies scale poorly while inconsistency-tolerating policies maintain close-to-linear scalability under random access pattern.

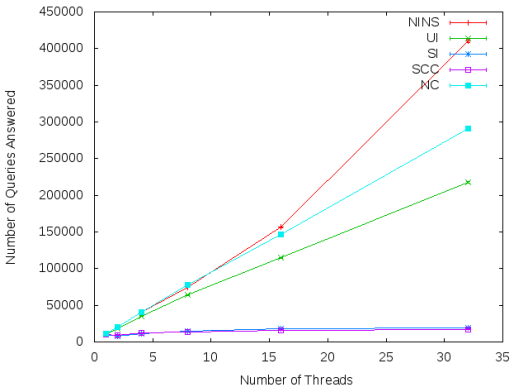
This experiment evaluates the throughput of the policies listed in Section 6.2 with a random access pattern where a worker randomly picks an entered cell to update or a computed cell to query. The prototype uses a linear pseudo-random number generator to pick the random cell to query or update. Therefore, the prototype executes the same action sequence if the same seed for the random number generation are used. The update rate (ur below the graphs in Figures 6.1 and 6.2) is the probability that a worker will execute an update rather than a query in the selected cell. The same experiment is run three times on both the simple data set and the manufacturer data set. In each run a different seed is used for the random generator. We report the average throughput along with the minimum and maximum throughput. The difference between minimum and maximum is so small that the interval is almost invisible in the figures. This experiment does not use any fresheners.

Figure 6.1 presents the results for the simple data set. We use up to 32 threads on this data set. We let the OLAP system run 6 seconds with different number of threads and report the throughput, measured as the number of queries answered, of them under various update rates.

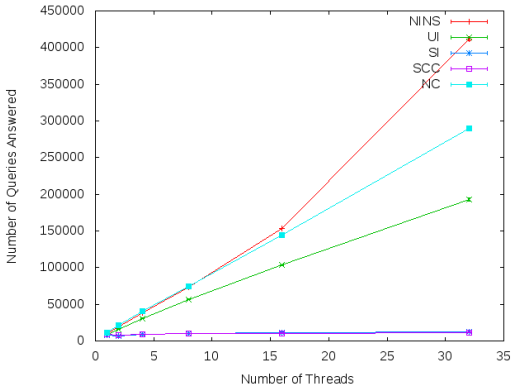
In Figure 6.1a, each policy has similar performance compared to the NC policy



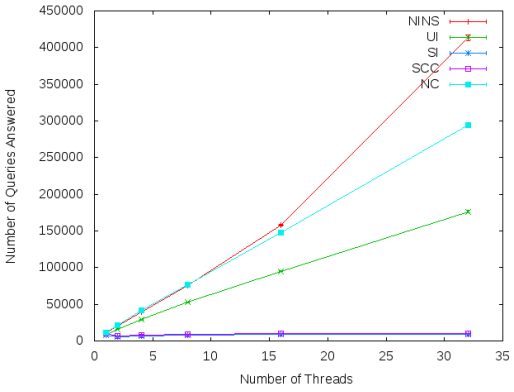
(a) ur = 0



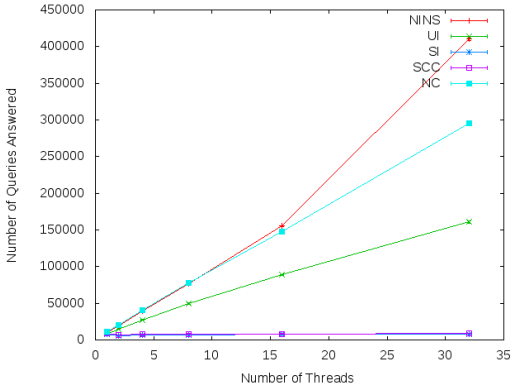
(b) ur = 5



(c) ur = 10



(d) ur = 15



(e) ur = 20

Figure 6.1: Simple Data Throughput

for thread count lower than 16 threads. In other words, they all have close-to-linear scalability when there's no update happening on the simple data set. The SCC policy and the SI policy scale poorly when using more than 16 threads. This poor performance is attributed to the poor scalability of the pthread read-write lock. With 32 threads, the OLAP system caches enough cells to accelerate future queries under the UI policy and the NINS policy and hence achieved the super linear scalability.

Figures 6.1b, 6.1c, 6.1d and 6.1e show the effect of the update rate on all the policies that use a cache system on the simple data set. The slightly worse performance of the UI policy compared to the NC policy is explained by the overhead for invalidation. Other policies with read-write lock cannot benefit from multi-threading at all because updates require the acquisition of a global write lock which stalls all other queries and updates. The NINS policy has almost no overhead compared to the NC policy for thread counts below 32. The performance benefits that the NINS policy brings by using cached values to accelerate computation are offset by the overhead of maintaining the cached cell set. Again, with 32 threads, the system cache enough cells under the NINS policy to achieve a super-linear scalability and to outperform the NC policy.

Figure 6.2 presents the results for the manufacturer data set. We use up to 16 threads on this data set. We let the OLAP system run for 60 seconds with different number of threads and report the throughput of them under various update rate.

Figure 6.2a also demonstrates the poor scalability issues for thread counts above 16 for policies with read-write lock. The UI policy in this figure is slightly worse than the NC policy. This phenomenon means that, in this experiment, the overhead of inserting cells into the set of cached cells with the UI policy is more significant than the benefit of using cached cells.

The results presented in Figure 6.2b indicate that policies with read-write lock still scale poorly. The SCC policy performs slightly better compared to the SI policy. In Figures 6.2c, 6.2d and 6.2e, the difference between the SCC policy and the SI policy increases as the update rate increases. The increase of the difference is due to the significant overhead of invalidation even without synchronization on some workload. With the SCC policy, updating is less time consuming compared to the SI policy, as we explained in Section 3.3. Therefore, the system holds the write lock for a shorter time. As a result, the SCC policy scales better than the SI policy.

The NINS policy performs better than all other policies with cache system but slightly worse than the NC policy in Figure 6.2a, 6.2b, 6.2c and 6.2d due to the overhead of maintaining a cached cell set. When the update rate increases, fewer queries are calculated and thus fewer cells are inserted into the cached cell set. Therefore, the overhead of maintaining the set decreases. With an update rate of 20 (Figure 6.2e), the invalidation-free approach finally catch up with the NC policy.

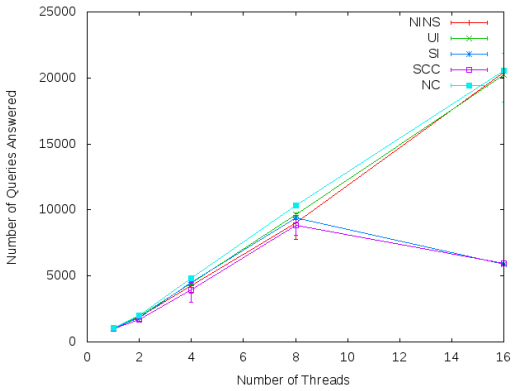
This experiment confirms the heavy overhead of synchronization technologies, such as read-write locks, on a multi-threaded cache system. In all these experiments the OLAP system queries and updates randomly selected cells. Therefore none of the approaches with cache performs better than the NC policy. This is result is expected because a random access pattern does not exhibit the locality characteristics that are necessary for a caching system to perform well. The UI policy and the NINS policy bring only slight overhead and thus dramatically outperform other approaches with a cache system. Unfortunately, these policies generate stale results that will be discussed later in this chapter.

6.4.2 Comparison between Inconsistency-Tolerating Policies

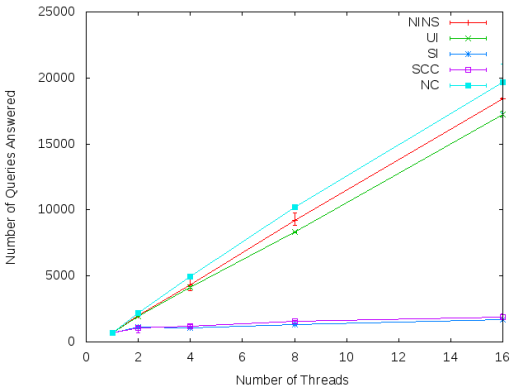
The results presented in this section confirm the expectation that the NINS policy provides better throughput while generating more stale query results compared to the UI policy.

The first experiment explores the throughput of inconsistency-tolerating policies under various workloads. This experiment mixes access patterns: a worker either picks a cell randomly from all data cubes or it picks a cell from a pre-selected set of hot cells to query. An input parameter to the experiment is the *Hot-Cell Probability* (HP) that specifies the probability that a worker would pick a cell from the hot-cell set to query.

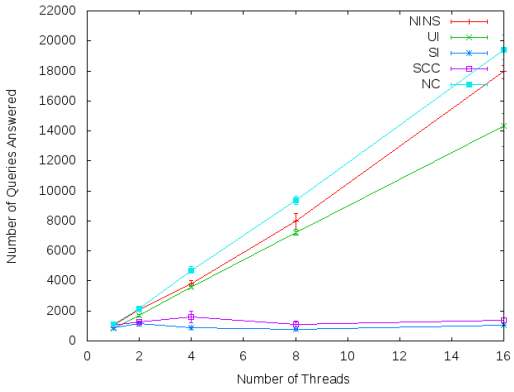
Different from previous experiments, in this experiment fresheners update stale values of cached cells. For all policies, the system uses fifteen workers and a single freshener. Additionally, different freshening policies are evaluated in this experiment. This experiment is only run on the manufacturer data set. For each evaluation point, the prototype executes the OLAP system for three minutes for the throughput experiment for ten minutes for the experiment that measures the query stale rate. After each minute statistics are collected. For box plots the distribution used to create each box is formed by the individual statistics collected at each of



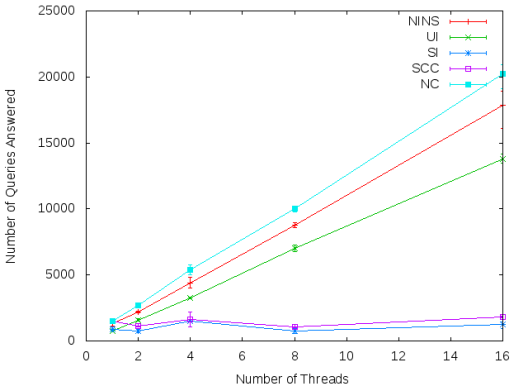
(a) ur = 0



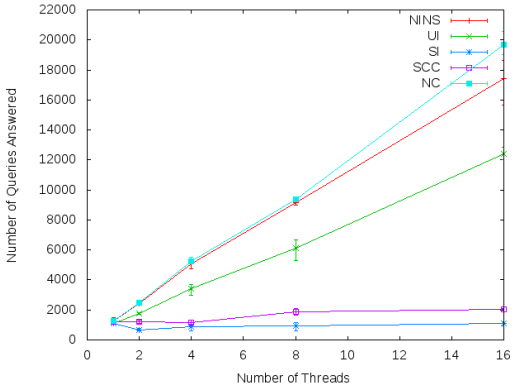
(b) ur = 5



(c) ur = 10



(d) ur = 15



(e) ur = 20

Figure 6.2: Manufacturer Data Throughput

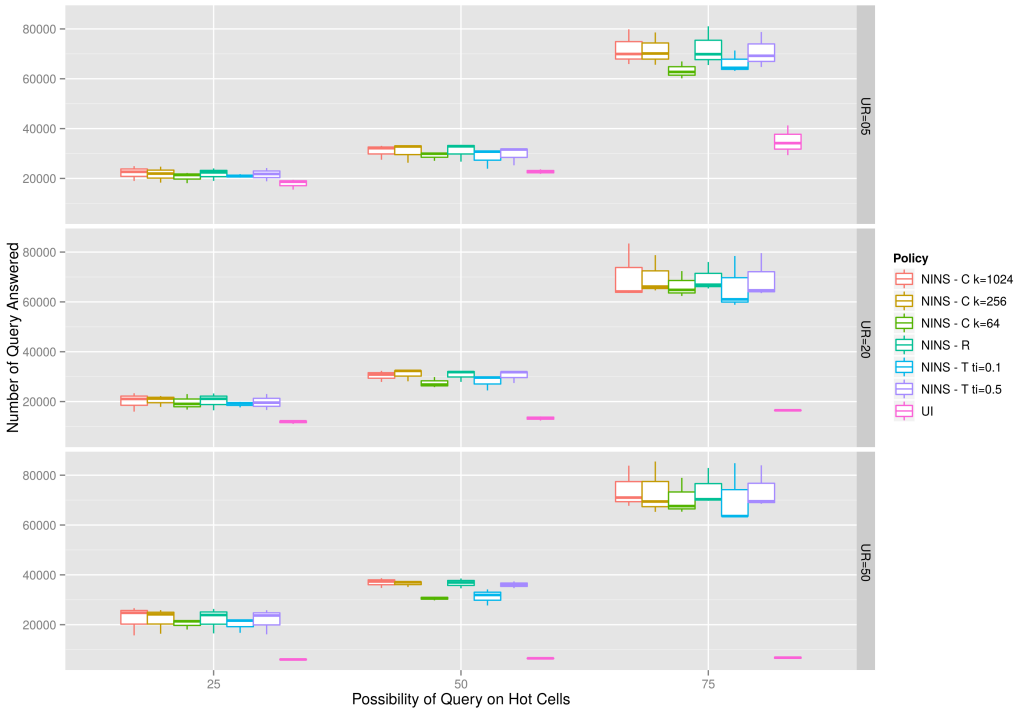


Figure 6.3: Distribution of number of queries answered for the manufacturer data set

these points. For instance, in Figure 6.3 each of the boxes is built from the number of queries answered measured at each of the statistic collection points during the execution.

Figure 6.3 confirms that the NINS policy answers more queries than the UI policy under various workloads especially when the update rate is high. This result confirms our expectation that the NINS policy further reduces the update overhead compared to the UI policy.

In Figure 6.3, both policies generate better throughput when HP is higher. This is obviously reasonable because the cache system performs well when the system is querying hot cells. The figure also shows that the UI policy throughput is more sensitive to the update rate. The overhead of invalidation during update dominates when the update rate is high and thus the UI policy does not benefit from higher HP.

The next experiment studies the level of inconsistency in the system, in terms of Query Stale Rate (QSR), when inconsistency-tolerating policies are used for the same set of workloads. This experiment confirms that the NINS policy results in higher levels of inconsistency compared to the UI policy.

Figure 6.4a shows the query stale rate of different policies under the same workloads used for the experiments in Figure 6.3. This figure shows nine plots organized on a 3×3 array. Each row corresponds to a value for the hot-cell probability (25, 50 or 75) and each column corresponds for a value for the update rate (5, 20 or 50). For this experiment, after each minute running, the system is paused to collect the query-stale-rate data and report in the graph. Therefore the time shown in the horizontal axis is not actual wall-clock time, but rather system running time because the pause time needed to collect the data is not shown. The graph shows that when combining the NINS policy with the round-robin freshening policy, the query stale rate gradually increases. This phenomenon confirms the conjecture, made in Section 4.3, that the freshener with round-robin freshening policy cannot efficiently control the inconsistency level especially when some cells are queried significantly more frequently than others.

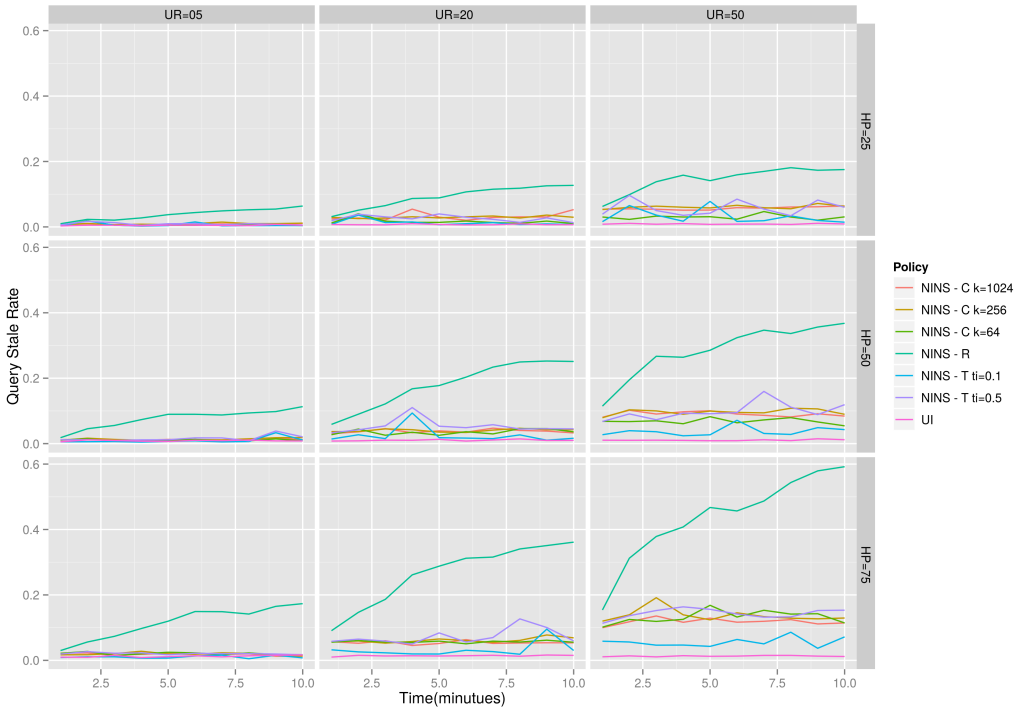
It is also clear that the stale rates for queries are higher under this policy when both the probability that a worker will pick a hot cell and the update rate are higher. It is obvious that the stale rate should increase as the update rate increases because the update may make a cached value stale. When the probability of selecting a hot cell is high, the OLAP system has more chance to use cached value to answer future query. Therefore, the system is more likely to generate incorrect query result because it is more likely to use a stale cached value.

Figure 6.4b presents a box-plot graph summarizing the results for all the values of update rates with a fixed hot-cell probability (HP).

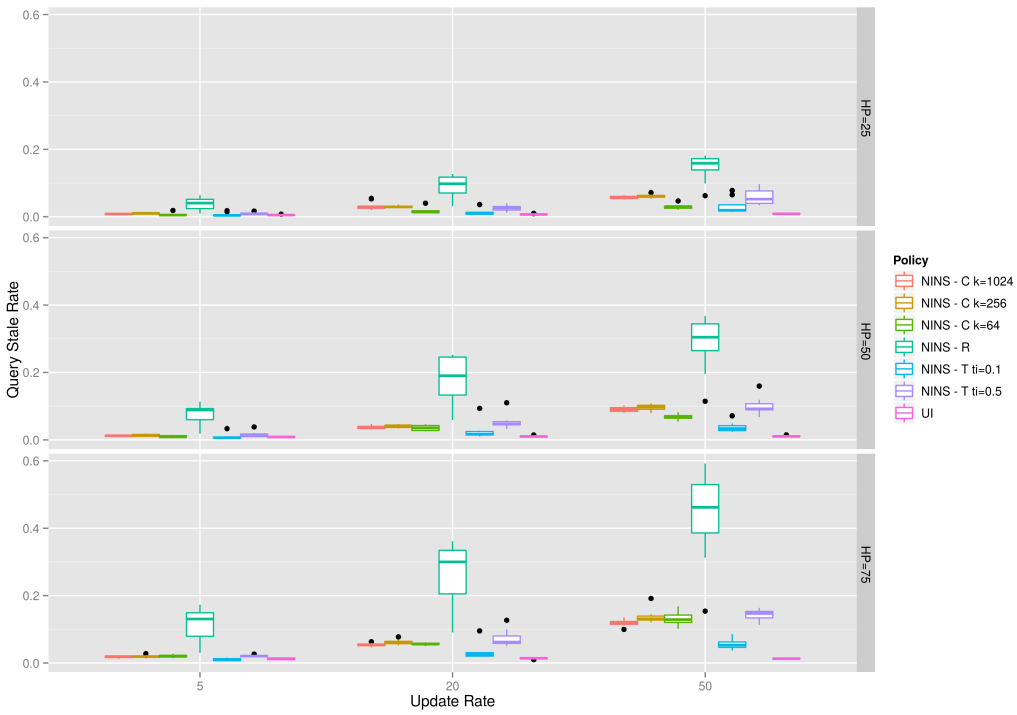
The NINS policy with a round-robin freshening policy has a high standard deviation because of the increasing query stale rate as time advances. For the NINS policy with an Access-Counter-Based MRQ freshening policy, the query stale rate increases as the update rate increases and k does not have any noticeable impact on the results. For the same policy with the Time-Based MRQ freshening policy, the query stale rate also increases when the update rate increases, but the time-interval choice affects the inconsistency level: a 0.1s time interval results in smaller query stale rate in comparison to a 0.5s time interval.

None of the freshening policies results in the NINS policy having as low a rate of stale queries as the UI policy when the update rate is high.

Overall, the NINS policy provides better throughput with more incorrect results compared to the UI policy. However, the MRQ freshening policy may not be the



(a) Actual Query Stale Rate variation over time



(b) Distribution of the Query Stale Rate

Figure 6.4: Query Stale Rate for the Manufacturer Data Set

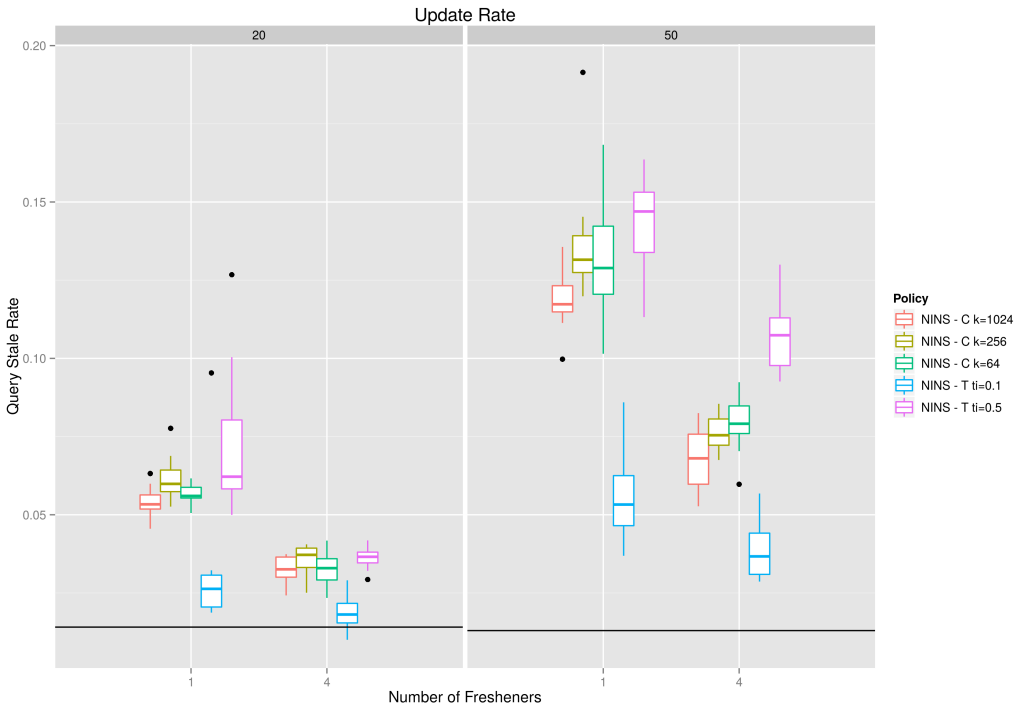


Figure 6.5: Distribution of Query Stale Rates for Manufacturer Data Set with HP = 75

best heuristic to select cached cells to re-calculate and invalidate. There's potential to explore more sophisticated heuristics to help the NINS policy achieve a better trade-off between inconsistency level and throughput.

6.5 Relationship between Number of Fresheners and Query Stale Rate

This section reports on an experiment that shows that, with the NINS policy, adding more fresheners has a positive but limited effect on reducing the query stale rate. The additional fresheners help reduce the inconsistency level but its effect is only significant when the inconsistency level is relatively high. A second experiment reveals that the NINS policy can answer more queries, even when it uses fewer work threads, compared to the UI policy.

These two experiments use four fresheners and twelve work threads and compare the result with the one in Section 6.4.2, which used one freshener and fifteen work threads. Additionally, we fix the probability of selecting a hot cell to 0.75, which results in a relatively high query stale rate.

Figure 6.5 clearly demonstrates the effect, on the query stale rate, of replacing work threads with fresheners. In each of these two box plots, the left one for an update rate of 20 and the right for an update rate of 50, a horizontal black line shows the level of the query stale rate for the Unsynchronized Invalidation policy with a single freshener. This query stale rate is the baseline for the comparison. For each of the policies, as the number of fresheners increases, the query stale rate drops. It is relatively efficient to add fresheners when the query stale rate is high. But the efficiency of additional fresheners drops when the query stale rate is low. For instance, for the NINS policy with the Time-Based MRQ freshening policy ti = 0.5s, the median of the query stale rate drops from 0.15 to 0.11 when the update rate is 50% but only drops from 0.06 to 0.04 when the update rate is 20% with 3 additional fresheners.

The Cached-Value Stale Rate (CVSR) was defined in Chapter 4 as the proportion of cached cells that contain a stale value. A low CVSR results in a low query stale rate and in a low freshener's efficiency because in this case a freshener spends most of its time rechecking correct cached value rather than updating stale ones. Therefore, the marginal benefits of query-stale-rate reduction for each additional freshener drops as the total number of fresheners increases. As a result, the ability to control the query stale rate through the adjustment of the number of fresheners is limited. At such a point a different policy, such as the UI policy, must be selected to further reduce the query stale rate.

The graph in Figure 6.6 is built in a similar fashion to the graph Figure 6.5, but it is reporting the throughput, instead of the query stale rate, for each of the policies with one and four fresheners. The total number of threads remains constant at sixteen. Therefore, with more freshener and fewer workers, the OLAP system answers fewer queries in the same amount of time. This finding is reasonable because workers are the only threads who can answer a query.

Nonetheless, the NINS policies still dramatically outperforms the UI policy on throughput. For instance, when the update rate is 50%, the NINS policy with Time-Based MRQ freshening policy (ti=0.01) uses 12 workers achieving 7.6x speed up compared to the UI policy which uses 15 workers.

The results for these two experiments reveal the possibilities and limitations to control the query stale rate by adjusting the number of fresheners with the NINS policy. Even when it uses more fresheners, the NINS policy cannot achive query

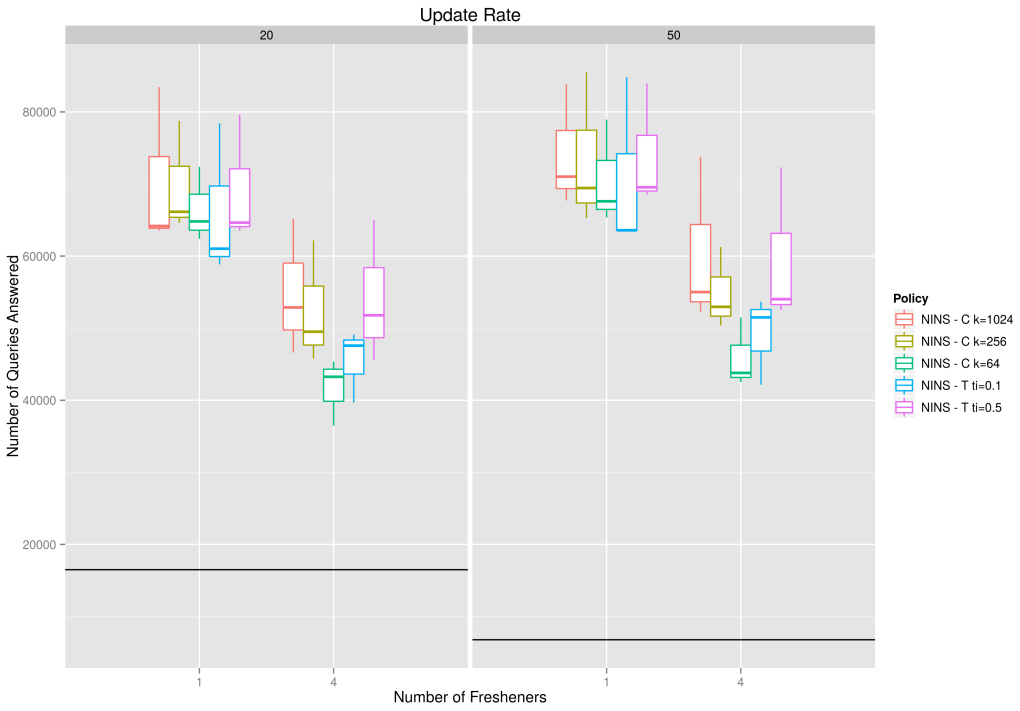


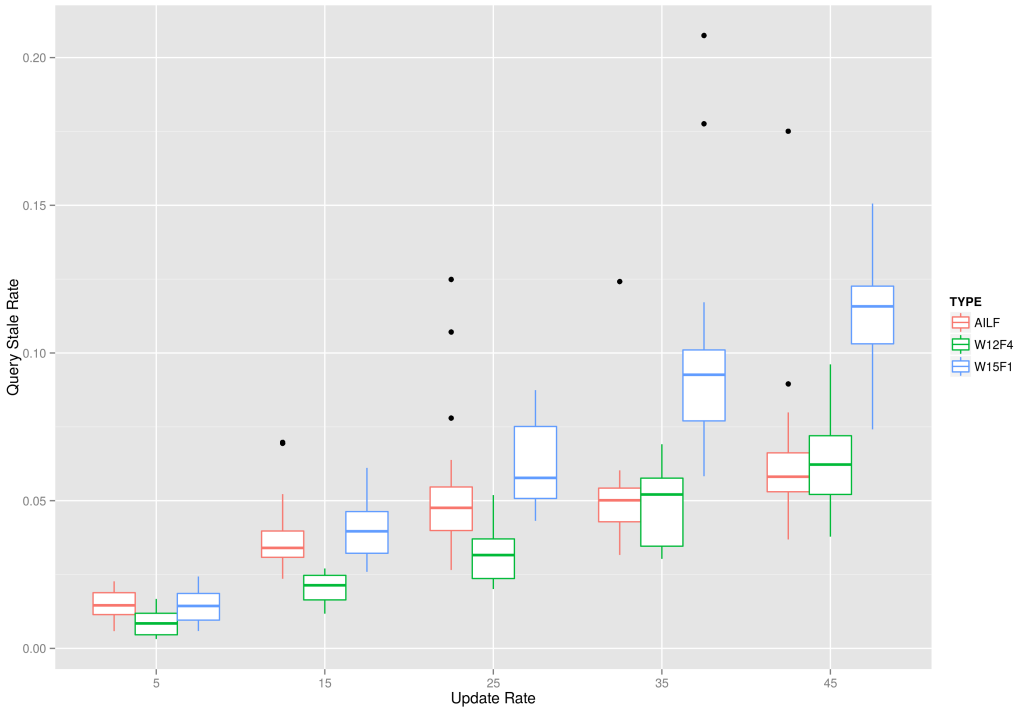
Figure 6.6: Distribution of Number of Queries Answered for Manufacturer Data Set with HP = 75

stale rates as low as the ones achieved by UI policy. However, the NINS policy does provide extra space to trade-off between the inconsistency level and the throughput in the system.

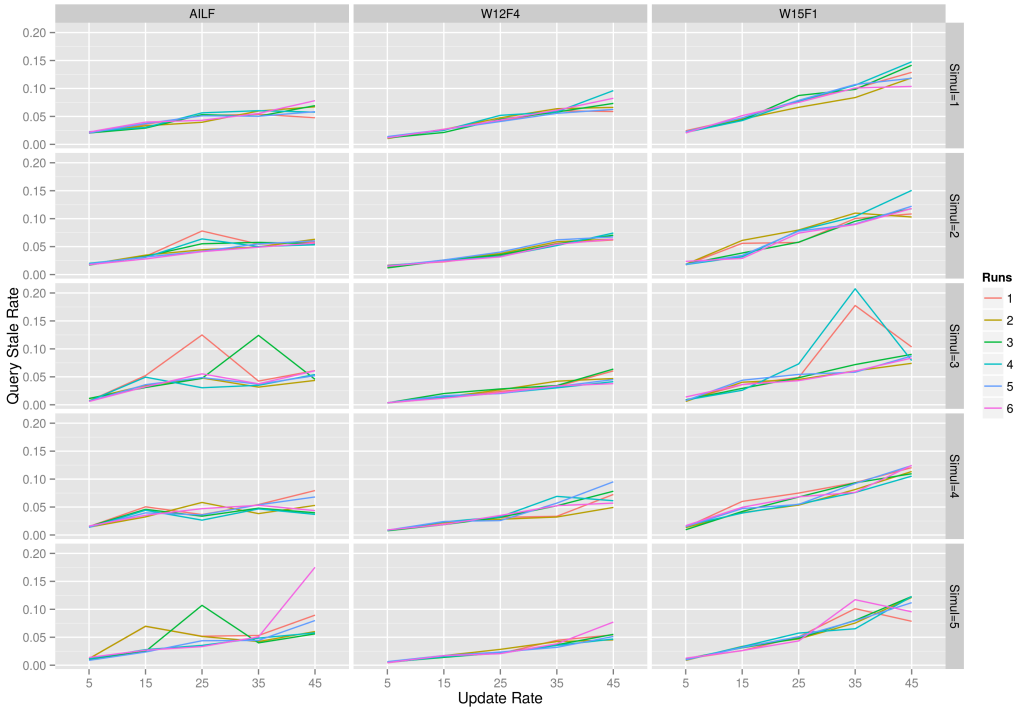
6.6 Effectiveness of AILF to Control Inconsistency Level

Chapter 5 introduces the Adaptive-Inconsistency-Level Framework (AILF) to dynamically adjust the number of fresheners according to the Cached-Value Stale Rate (CVSR). This section presents the results of an experiment that demonstrates the ability of AILF to control the inconsistency level in the long run and its limited ability to reduce short-run inconsistency-level variations.

This experiment uses the NINS policy with the Time-Based MRQ freshening policy (ti=0.5s). The goal is to compare AILF with non-adaptive strategies that keep the number of fresheners constant. In all strategies used in the experiment the total number of threads is sixteen. The experiment compares AILF is compared with a strategy that uses a single freshener (W15F1 in the graphs) and with another that operates with four fresheners (W12F4). In this experiment, every minute the update



(a) Distribution of Query Stale Rates



(b) Query Stale Rates variations over time

Figure 6.7: Query Stale Rates for Manufacturer Data Set with HP = 75 and ti=0.5s

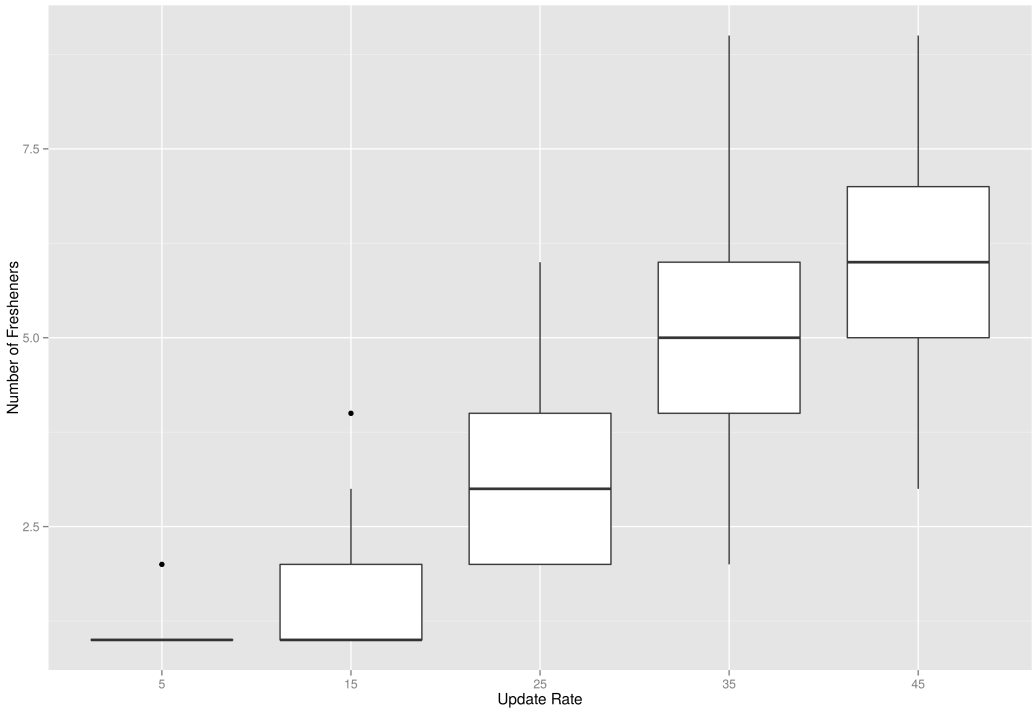


Figure 6.8: Number of Fresheners used by AILF

rate increases by 5%, starting at 0.5 and going up to 0.45. The experiment consists of five simulations with different random-number-generator seeds. Each simulation is run six times. The results are reported win the boxplot graph in Figure 6.7a and in the collection of line graphs in Figure 6.7b. In the array of graphs in Figure 6.7b, each column represents a different strategy and each row is for a different simulation.

The results shown in the graph of Figure 6.7a indicate that AILF reduces the query stale rate in comparison to the strategy with a single freshener, and maintains it below 5%. But with AILF, we observe several outliers especially when UR is 25. To understand these outliers, we list the query stale rate for each simulation and each run in Figure 6.7b. There outliers appear only in particular simulations (3 and 5) and are caused by variations in the query stale rate. Such vibrations also appear in the strategy with a single freshener. Such vibration can easily caused by keep updating and querying on related cells in a short period. Unfortunately, the AILF cannot efficiently reduce such vibration. AILF adjusts the number of fresheners according to the history. Therefore it cannot react to an abrupt changes on workload in time.

Figure 6.7b also shows that AILF is not more effective than the strategy with four

fresheners regarding to query stale rate. Figure 6.8 shows the number of fresheners used by AILF. AILF uses less than four fresheners when the update rate is less than 25. In other words, it allocates more workers to execute queries and updates. When the update rate is 25, the AILF also uses less than four freshener in most cases. When the update rate is larger than 25, the number of fresheners increases much faster. There is only a marginal benefit to adding freshener as the total number of fresheners increases. A future study could investigate a strategy that stop adding freshener when its benefit becomes indistinctive.

From this experiment, it is clear that the current AILF is helpful to keep the query stale rate stable in the long run but can hardly smooth the vibration.

6.7 Conclusion

The experiments reported in this chapter quantify the improvements that come from the use of a cache system in a real-time OLAP system under specific workloads. The results also reveal the poor scalability of the strategy that maintains the consistency of cached values through the use of synchronization. Next, the experimental evaluation provided support to the proposal that the elimination of synchronization improves the performance of the OLAP system while resulting in an acceptable level of inconsistency for a manufacturer data set. Moreover, an innovative policy, the NINS policy, allows for trading consistency level for higher throughput, *i.e.* if higher levels of inconsistency are tolerable the policy can increase the throughput of the system. Finally, the AILF can be used to control the inconsistency level.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

This thesis explores the cost of synchronization in real-time OLAP systems with updates, and proposes new policies to trade consistency for lower synchronization overhead.

A cache system with read-write lock can only bring benefit to the OLAP application with very limited set of workload due the heavy overhead during the invalidation procedure. Elimination of synchronization during invalidation, and relying on fresheners to maintain acceptable level of inconsistency (proposed by Ungar), reduces the overhead and dramatically improves the scalability of the system. Hence, this approach significantly broaden the scope of suitable workloads [30]. This thesis takes this approach one step further and proposes the complete elimination of invalidation. The idea is to rely on fresheners to both refresh stale values of cached cells and perform invalidation. The experimental results indicate that this new approach further reduces the update overhead. However it may introduce a high inconsistency level that may not be acceptable for most applications.

We propose to collect inconsistency level information, using a newly defined Cache-Value Stale Rate metric, while freshening cached cells. Based on this information, we proposed the Adaptive Inconsistency Level Framework (AILF) that dynamically adjusts the number of fresheners to control the inconsistency level. The experimental results indicate that AILF is able to controlling the inconsistency level in the long term but has limited ability to deal with short-term variations in the inconsistency level.

This preliminary evaluation using this new prototype signals to significant promise for the proposed scheme. A key investigation for future research is to use actual

recorded workloads from the operation of large organizations to study the tradeoff between synchronization costs and inconsistency level. The new framework to measure this tradeoff presented in this thesis should be useful for such studies.

7.2 Future Work

This thesis uses a portion of a manufacturer data set obtained from a global manufacturing company to evaluate the throughput and inconsistency level on different consistency policies in a real-time OLAP system that processes updates. This initial prototype is not able to load the entire data set that we received from the company. Moreover, most of the experiments reported in this thesis use a random-access workload and a limited query set workload. These workloads are certainly different from workloads that would be produced by the actual use of this data set in the daily operation of the company. Future experimental evaluations should use workload that more closely simulate the real workload and should also use the whole manufacturer data set.

Currently, all invalidation policies iterate over all computed cells that may be affected by a changed entered cell in the cube in order to discover the cells that need to be invalidated. As indicated in Section 2.4.3, it is possible to group the cells and record the relations among cell groups using a dependency graph to achieve better performance.

In the current prototype all the fresheners apply the same freshening policy (either round robin or MRQ) and they do not cooperate efficiently with each other. An interesting line of research is to investigate the effect of allowing different fresheners to focus on different groups of cells to avoid repeatedly freshening the same cell in a short time interval. Such cooperation strategy may help to improve the freshening efficiency.

Alternative adaptation strategies can be investigated for the AILF. For instance, the AILF should be allowed to add or remove more than one freshener at a time according to the current status. With more sophisticated adjustment strategies, the AILF may be able to maintain a much more flat QSR curve compared to the results presented in Section 6.6.

Finally, the AILF should be able to do more than adjusting the number of fresheners. With the current run-time information, including the inconsistency level information, it should be able to decide: 1) what's invalidation policy to apply, 2)

how many fresheners are needed, 3) how should each freshener select cells to fresh?

Bibliography

[1] Accessing olap using asp.net. http://www.aspfree.com/c/a/ms-sql-server/accessing-olap-using-asp-net/, 2014. [Online; accessed 30-June-2014].

[2] datanova business intelligence. http://www.datanovasoftware.com/propane_en.html, 2014. [Online; accessed 30-June-2014].

[3] Ibm cognos tm1. http://www-03.ibm.com/software/products/en/cognostm1/, 2014. [Online; accessed 30-June-2014].

[4] iccube. http://www.iccube.com/, 2014. [Online; accessed 30-June-2014].

[5] Rolap wiki page. http://en.wikipedia.org/wiki/ROLAP#Advantages_of_ROLAP, 2014. [Online; accessed 30-June-2014].

[6] Martín Abadi, Butler Lampson, and Jean-Jacques Lévy. Analysis and caching of dependencies. *ACM SIGPLAN Notices*, 31(6):83–91, 1996.

[7] Umut Acar, Guy Blelloch, Matthias Blume, Robert Harper, and Kanat Tangwongsan. A library for self-adjusting computation. *Electronic Notes in Theoretical Computer Science*, 148(2):127–154, 2006.

[8] Umut A Acar, Guy E Blelloch, and Robert Harper. *Adaptive functional programming*, volume 37. ACM, 2002.

[9] Surajit Chaudhuri and Umeshwar Dayal. An overview of data warehousing and olap technology. *ACM Sigmod record*, 26(1):65–74, 1997.

[10] Surajit Chaudhuri and Kyuseok Shim. Including group-by in query optimization. In *VLDB*, volume 94, pages 354–366, 1994.

[11] Li Chen, Wenny Rahayu, and David Taniar. Towards near real-time data warehousing. In *Advanced Information Networking and Applications (AINA), 2010 24th IEEE International Conference on*, pages 1150–1157. IEEE, 2010.

[12] Edgar F Codd, Sharon B Codd, and Clynch T Salley. Providing olap (on-line analytical processing) to user-analysts: An it mandate. *Codd and Date*, 32, 1993.

[13] Umeshwar Dayal. Of nests and trees: A unified approach to processing queries that contain nested subqueries, aggregates, and quantifiers. In *Proceedings of the 13th International Conference on Very Large Data Bases*, pages 197–208. Morgan Kaufmann Publishers Inc., 1987.

[14] Richard A Ganski and Harry KT Wong. Optimization of nested sql queries revisited. *ACM SIGMOD Record*, 16(3):23–33, 1987.

[15] Matthew Hammer, Khoo Yit Phang, Michael Hicks, and Jeffrey S Foster. Composable, demand-driven incremental computation.

[16] Allan Heydon, Roy Levin, and Yuan Yu. Caching function calls using precise dependencies. *ACM SIGPLAN Notices*, 35(5):311–320, 2000.

[17] Won Kim. On optimizing an sql-like nested query. *ACM Transactions on Database Systems (TODS)*, 7(3):443–469, 1982.

[18] Jukka Kiviniemi, Antoni Wolski, Antti Pesonen, and Johannes Arminen. Lazy aggregates for real-time olap. In *DataWarehousing and Knowledge Discovery*, pages 165–172. Springer, 1999.

[19] James L McKenney and Morton M Scott. *Management decision systems: computer-based support for decision making*. Harvard Business School Press, 1984.

[20] Sasa Misailovic, Deokhwan Kim, and Martin Rinard. Parallelizing sequential programs with statistical accuracy tests. *ACM Transactions on Embedded Computing Systems (TECS)*, 12(2s):88, 2013.

[21] Sasa Misailovic, Stelios Sidiroglou, and Martin C Rinard. Dancing with uncertainty. In *Proceedings of the 2012 ACM workshop on Relaxing synchronization for multicore and manycore scalability*, pages 51–60. ACM, 2012.

[22] Torben Bach Pedersen and Christian S Jensen. Multidimensional database technology. *Computer*, 34(12):40–46, 2001.

[23] Daniel J Power. A brief history of decision support systems. *DSSResources. COM, World Wide Web, http://DSSResources. COM/history/dsshistory. html, version*, 4, 2007.

[24] William Pugh and Tim Teitelbaum. Incremental computation via function caching. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 315–328. ACM, 1989.

[25] Lakshminarayanan Renganarayana, Vijayalakshmi Srinivasan, Ravi Nair, and Daniel Prener. Programming with relaxed synchronization. In *Proceedings of the 2012 ACM workshop on Relaxing synchronization for multicore and manycore scalability*, pages 41–50. ACM, 2012.

[26] Ricardo Jorge Santos and Jorge Bernardino. Real-time data warehouse loading methodology. In *Proceedings of the 2008 international symposium on Database engineering & applications*, pages 49–58. ACM, 2008.

[27] Praveen Seshadri, Hamid Pirahesh, and TY Cliff Leung. Complex query decorrelation. In *Data Engineering, 1996. Proceedings of the Twelfth International Conference on*, pages 450–458. IEEE, 1996.

[28] Ralph H Sprague Jr and Eric D Carlson. *Building effective decision support systems*. Prentice Hall Professional Technical Reference, 1982.

[29] Erik Thomsen. *OLAP solutions: building multidimensional information systems*. John Wiley & Sons, 2002.

[30] David Ungar, Doug Kimelman, and Sam Adams. Inconsistency robustness for scalability in interactive concurrent-update in-memory molap cubes. *Inconsistency Robustness*, 201, 2011.

[31] Colin White. Intelligent business strategies: Real-time data warehousing heats up. *DM Review*, 2002.

[32] Yihong Zhao, Prasad M Deshpande, and Jeffrey F Naughton. An array-based algorithm for simultaneous multidimensional aggregates. In *ACM SIGMOD Record*, volume 26, pages 159–170. ACM, 1997.