Aggregate Queries in NoSQL Cloud Data Stores

Master's Thesis, PDCS
Submitted to the Department of Sciences, Vrije Universiteit, Amsterdam,
The Netherlands

Plamen Nikolov (2000229)
August 2011

Guillaume Pierre
Principal Advisor

Abstract

This thesis work describes the design and implementation of an aggregate view maintenance mechanism for web applications in the Cloud. Maintaining consistent views and ensuring fault and partition tolerance is generally hard as the CAP theorem postulates that these three properties cannot hold simultaneously. Nevertheless, web application transactions are often characterized by short-lived transactions touching only a few data items. Specific examples are shopping cart updates, user name and password look-ups, and online visitor statistics. The small sized updates allow for an efficient aggregate view maintenance solution based on incremental updates via change tables. The change table mechanism can be readily implemented at a transaction coordinator node which can carry out the entire computation because of the small-sized view change tables.

This thesis elaborates on using the two-phase commit protocol and simple equi-join queries to implement a synchronous and asynchronous version of the change table algorithm. In this work we will argue that the heavy workloads introduced by the synchronous approach can be decreased by relaxing the view freshness requirements. Thus, the performance of the proposed mechanism can be controlled by exploiting the whole consistency range from immediate updates to deferred refreshing which can be carried out at arbitrary time intervals.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

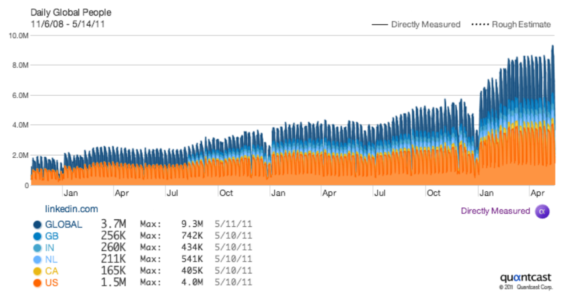
1.1 From Relational Databases to NoSQL

According to Moore's law transistor density doubles every two years while semiconductor prices decrease steadily [19]. The exponential growth in computer power and cost-performance ratio has led to an increased availability of commodity computers and widespread usage of cluster and on-demand computing on the Cloud. The shift from expensive state-of-the-art machines to inexpensive commodity hardware necessitates rethinking web application models from resource provisioning to storage engines in order to allow for user base expansions and handling hardware failures in a distributed environment. With the increase of computational power, the application's user base grows as well (Figure 1.1), challenging web services with millions of users demanding low response times and high availability. As in reality users increase asymptotically faster than transistor density, system architect wizards have to resort to increasingly sophisticated tricks in order to address the scalability issues in the data storage tier.

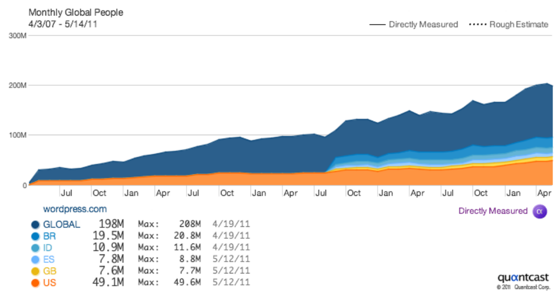
Generally, database scalability can be achieved by either vertically or horizontally scaling the database tier:

- **Vertical Scalability:** When scaling vertically, the database tables are separated across different database instances on potentially distinct machines so that each server is assigned a specific task. While this approach results in efficient I/O load-balancing, vertical scalability depends on the presence of logically separable components of the database and the ability to constantly upgrade the existing hardware;

- **Horizontal Scalability:** When scaling horizontally, the database structure remains unchanged across all the database instances. Achieving horizontal partitioning requires a consistent maintenance between the different instances and their replicas, efficient load balancing, and I/O aware query algorithms so that the data transfer



(a) LinkedIn Traffic



(b) WordPress Traffic

Figure 1.1: LinkedIn and WordPress Platform monthly traffic measured in people per month (Source: Quantcast.com)

latency is minimized. Theoretically, the achieved database speed-up by horizontal scaling is proportional to the number of newly added machines.

For Cloud-based web-applications, vertical scalability is not a viable solution as it involves adding new resources to a single node in the system. While the prices of off-the-shelf components have radically decreased over the last several years, there is a limit to the number of CPUs and memory a single machine can support. Typically, systems are scaled vertically so that they can benefit from virtualization technologies; a common vertical scalability practice is placing different services and layers on separate virtual machines to facilitate migration and version management.

One of the biggest challenges facing web applications is not the lack of computational power but efficiently and resiliently processing a huge amount of database query traffic [22]. More than 30 years old, RDBMSs represent the perfect storage solution on high-end multi-core machines equipped with enormous hard drives. With their impressive feature set, transaction management, and query capabilities relational database solutions are able to handle nearly any imaginable task. However, problems start arising once these databases have to become distributed in order to handle the more demanding traffic; coming from the era of mainframes, they were never designed to scale. Below are listed only some of the issues causing relational databases to lose the lead in large scale web applications:

- Behemothian data volumes require applying a heavy data partitioning scheme across a huge number of servers, also referred to as sharding[1], leading to a degraded performance of table joins;

[1] A popular distributed database design is horizontal partitioning in which the database rows are distributed across different partitions. Each partition belongs to a shard which might be located on a different database server.

- Systems with heavyweight transaction management cannot handle efficiently concurrent queries;
- Relational databases cannot handle efficiently data aggregation from large shard volumes due to high communication costs.

Many of the described scalability issues can be addressed by relaxing the consistency requirements and dropping the relational schema. This argument is especially pertinent to web applications as they exhibit a different data access behavior than enterprise data mining platforms. A typical model-view-controller web application uses a small subset of SQL with fine-grained data access patterns and performs very few writes compared to the number of read operations. Thus, the data model can be readily expressed as key-value pairs or simple document corpora which can be accommodated by data stores such as Google's BigTable [8], Amazon's SimpleDB [3], and open-source projects such as HBase and Cassandra. These and other similar storage solutions go under the umbrella name of NoSQL.

1.2 Scalable and Consistent R/W and Equijoin Queries in the Cloud

Currently most NoSQL databases provide extremely limited functionality as they do not support atomic updates, join, and aggregate query capabilities. Often when consistency is not a key requirement, porting the storage tier of an application from a relational database to a non-relational solution such as Hadoop only requires denormalizing the data model [7]. This is the case for applications using "approximate" information[2] (e.g. approximate number of a website page views). On the other hand, there are many applications which cannot use a weak data consistency model. For example, overselling a physical item in an online store is a highly undesirable event at best.

Providing strong consistency in non-relational stores is generally a hard problem. Intuitively, the distributed nature of the data results in a higher cost to synchronize all changes made by a transaction into a known single state. More formally, these challenges are captured by Brewer's theorem postulating the impossibility for a distributed system to simultaneously provide consistency, availability, and partition tolerance [10]. Nevertheless, web applications exhibit certain data access patterns which ease the strong consistency problem.

The CloudTPS middleware introduces a novel approach for achieving strong consistency in NoSQL by taking advantage of the short-lived, relatively small queries characteristic to web applications [23]. The proposed framework uses multiple transaction managers to maintain a consistent copy of the application data to provide developers with a means

[2]The database operations in the "approximate" reads scenario are not serializable. The final outcome is not equal to the outcome of a serial execution of the operations.

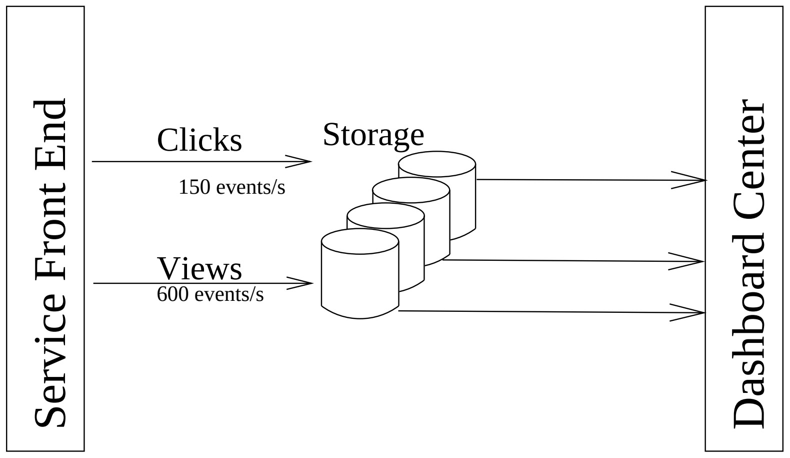


Figure 1.2: Data Flow within a Web Application

to create web applications by following common engineering practices instead of circumnavigating the peculiarities of NoSQL. For example, the TPC-W benchmark [18] can be implemented on CloudTPS with a minimum number of modifications [23]. However, CloudTPS lacks the implementation of an important query class–data aggregation.

1.3 Motivation

As data sets and user bases grow larger, so does the need for aggregate queries in the Cloud. For example, Wikipedia and the Wordpress platform already use extensively MIN-MAX queries. E-commerce web applications often provide real-time dashboards providing functionality from aggregating the contents of a shopping cart to displaying sophisticated analytics about clicks, impressions, or other user events. Figure 1.2 shows a simplified scheme of a large scale web application providing real time analytics for user events. With several hundred events per second, performing real time data aggregation using a relational database is an intimidating task due to the large number of write operations and reads spanning numerous shards.

Formally, data aggregate queries are defined as functions where the values of multiple rows are grouped together according to some criteria–a grouping predicate–to form a single value of more significant meaning. By far, the most popular aggregate functions in web applications are count, sum, average, minimum, and maximum over the result of a select-project-join (SPJ [3]) query:

- *Count()* returns the number of rows;

[3]An SPJ query represents a single SELECT-FROM-WHERE block with no aggregation or subqueries

- *Sum()* returns the sum of all values in a numeric column;
- *Average()* returns the average of all values in a numeric column.

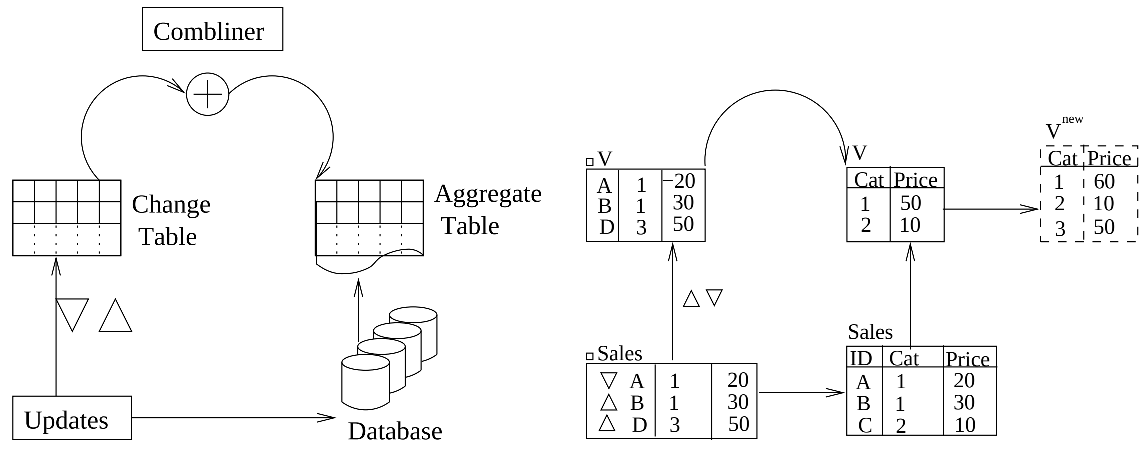
Currently none of the existing approaches for computing aggregate functions in the Cloud targets web applications using a generic data store. Instead, they exploit knowledge about the data model, compute the result offline, or take advantage of the functionality present in a specific data store; obviously, none of these approaches is applicable for the service described on Figure 1.2.

1.4 Aggregate Queries for Large-Scale Web Applications

Many web applications such as Wikipedia, the WordPress platform, and the dashboard application from Figure 1.2 use short-lived transactions spanning only a few rows in the underlying data store. For example, in the dashboard application, each click results in the update of the rows corresponding to the page being viewed. The relatively small size of typical web application transactions allows the efficient maintenance of *summary tables* containing the results of predefined aggregate queries. By using summary tables, incoming aggregation requests are served by simply looking up the result in the appropriate summary table. Because the aggregation work has been delegated to the mechanism updating the aggregate summary tables, the data store can treat all incoming aggregate queries as simple read-only operations and take advantage of the underlying horizontal scalability. Intuitively, the data in the aggregate summary table can be incrementally maintained; for example, inserting a row in a table participating in a *COUNT* query may result in increasing the aggregate result. This approach is shown on Figure 1.3a:

1. Initially, the summary tables are directly computed from the data available in the application's tables;
2. As updates, deletions, and insertions are coming in, the data store commits the modifications and computes *change tables* which "describe" how the updates affect the summary tables; the new value of a summary table after an update is a function of the old value and of the data present in the change table;
3. The data store uses the change tables to update the aggregate summary tables.

Figure 1.3b illustrates the change table approach for aggregate summary tables. Initially, the *Sales* table contains three products which have been aggregated by category in table V (View). At some point, the data store receives a transaction containing one insertion and two deletions and computes the corresponding change table $\Box V$. As deletion operations may decrease and never increase the number of rows participating in the aggregate result, the numerical attributes participating in the aggregation have been negated; as shown on



(a) Data Flow with Summary Tables (b) Change Table Example

Figure 1.3: Aggregate View Maintenance via Summary Tables

the diagram, A's price in $\square V$ has been set to -20. It is easy to see how $\square V$ describes the modifications which have to be applied to V in order to make the summary table become consistent with the updates: each row from $\square V$ is matched with rows from V according to the *Category* attribute and upon a successful match, the *Price* attribute is added to the aggregate result. A natural observation is that the described "matching" operation is equivalent to an inner join. Section 4 will provide a formal description of this approach.

The usage of change tables to maintain aggregate views has been studied extensively in the relational database world [1], [12], [14], [20], [21]. The strongly consistent read-only and read/write transactions in CloudTPS make the CloudTPS middleware suitable for implementing the summary table mechanism for aggregate queries. This thesis extends the change table technique to NoSQL data stores by adapting the view maintenance algorithm for usage in CloudTPS. The approach shown on Figure 1.3a can be naively implemented by merging the change and aggregate tables using only one server in the distributed environment. Unfortunately, this centralized solution would require shipping the entire aggregate summary table and result in a high bandwidth usage and latency penalty.

A data store's scalability can be captured informally by Amdahl's law [4]. If P is the portion of a program that can be parallelized, the maximum achievable speedup by N processors is $\frac{1}{(1-P)+\frac{P}{N}}$. Intuitively, the naive solution to the aggregate view maintenance problem can be naturally improved by parallelizing the combiner process presented on Figure 1.3a. As the aggregate view and change tables are automatically partitioned and replicated by the CloudTPS middleware, in the following sections we will concentrate on two view maintenance algorithms which generate the aggregate change tables and consistently update their corresponding views.

1.5 Thesis Structure

Before proceeding with the aggregate query algorithms and their implementation, we will start in Chapter 2 with a motivating example which will be used throughout this text. Chapter 3 will introduce several existing solutions to the aggregate problem and their shortcomings in the context of Cloud-based web applications. Next, Chapter 4 will review the theory behind the proposed solutions. Chapters 5 and 6 will deal with the actual aggregation algorithm implementation details and evaluation. Finally, Chapter 7 will recapture the aggregate view maintenance approach and provide a conclusion.

Chapter 2

Background

2.1 Motivating Example

As discussed in the introduction section, large-scale web applications exhibit a different behavior than data warehouse applications. Unlike large-scale enterprise OLAP (Online analytical processing) platforms, web applications have short-lived queries which often span only a few tables. The following example discusses a simple web application which will be used for illustrating some major points throughout this thesis.

Figure 2.1 depicts the architecture of the accounting component of a simple e-commerce application which displays products and receives revenue for product views and clicks. Formally, the front end generates "click" and "view" events whenever a product is clicked and viewed respectively and updates a database. The application's database has the following schema:

- *Events(ID, Timestamp, ProdID, ShopID, Type, Cost)* stores all user events. Each event is represented by a unique *ID*, a *Timestamp* indicating the time at which it occurred, the product's and shop's IDs *(ProdID, ShopID*), the event *Type*, and the *Cost* which will be charged to the shop's account.

- *Categories(Id, Name, Parent)* organizes the available products into categories–books, videos, etc. Each category is represented by a unique *ID*, *Name* (e.g. "books" and "videos"), and *ParentID* used for generating a category tree.

- *Products(ID, ShopID, CatID, Name)* stores all available products. Each product is represented by its unique *ID*, the shop offering the product (*ShopID*), its category (*CatID*), *Name* and *Description*, and cluster (*ClusterID*) grouping products which are instances of the same product (for example, two books with the same ISBN would have different IDs and equal ClusterIDs).

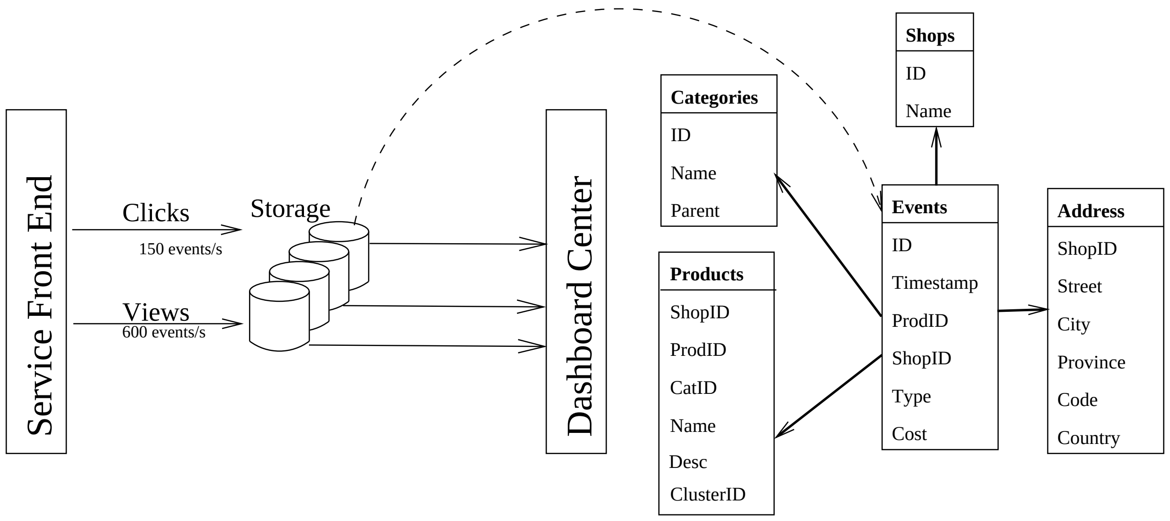


Figure 2.1: Web Application Accounting for User Clicks and Page Views

- Shops(*ID*,*Name*) stores the shops advertising products via the service; each shop is uniquely identified by its *ID*.
- *Address(ShopID, Street, City, Province, Code, Country)* stores the physical location of each shop advertising products.

The web application features a "Dashboard Center" allowing various analytics queries. For example, an end-user can examine the number of clicks and views a given product or category received and the resultant revenue. If the back-end storage solution is a relational database, finding the cost a specific merchant accumulated for the product views belonging to some category can be accomplished by issuing the following query:

Listing 2.1: SQL-1

```
SELECT Sum(cost) AS Cost FROM Display ,Merchant ,Categories
WHERE Display.merchantId=Merchant.merchantId AND Categories.
    catId=Display.catId AND Categories.catId='books' AND
    Merchants.merchantId='Kinokuniya'
```

A typical query plan of the SQL query presented above is shown on Figure 2.2. In order to find the cost accumulated by the "Kinokuniya" shop in category "books", the database scans the *Merchant* and *Categories* tables in order to retrieve the respective unique IDs. Next, the first join operator, denoted by $\bowtie_{ID}$, outputs all rows from the *Display* table corresponding to products sold by "Kinokuniya". The second join operator, $\bowtie_{category}$, yields all products belonging to the "Books" category. Finally, the result is summed over the *Cost* column.

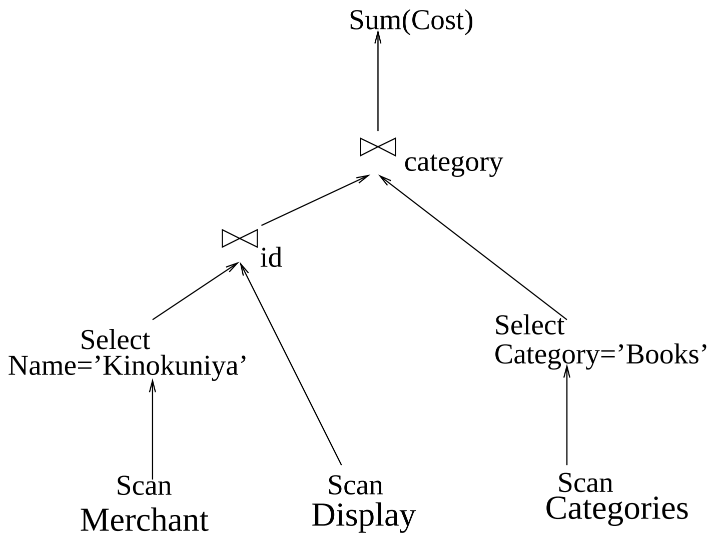


Figure 2.2: SQL Query Plan

The *Scan* and *Join* operations on Figure 2.2 exhibit data access patterns which have to be considered when looking into the performance of a Cloud implementation of the "Dashboard Center" web-application. In particular, the scalability of an application is largely influenced by the frequency at which it accesses different data items as well as by the physical data distribution across multiple hosts. The *Scan* operator makes only one pass through the table, emitting blocks matching the specified predicate. Moreover, the number of rows scanned is proportional to the table size. The asymptotic performance of the *Join* operator depends on the underlying database implementation. For example, a naive implementation would compare each tuple from the outer relation with each tuple in the inner relation, resulting in a quadratic cost of the comparisons required. A more sophisticated approach, hash joining, requires reading the entire inner relation and storing it into a hash table which is queried for each tuple present in the outer relation. Nevertheless, in both cases each table row has to be accessed at least once, leading to complications in cases where the same query has to be repeatedly executed in a distributed environment, as we shall see shortly. Table 2.1 estimates the number of row accesses per operation in order to compute the aggregate query on Figure 2.2.

It is easy to see that the query shown on Listing 2.1 conforms to web-application query properties. First, it is short-lived as the query results needs to be delivered in "real-time". Second, unlike enterprise data warehouse aggregations, the aggregation query plan does not involve sifting through terabytes of data.

The "Dashboard Center" described above is a typical relational database application with data flowing from an online transaction processing database into the data warehouse on an hourly basis. As evident from Table 2.1, even a simple aggregate query such as

Operation	Accessed Items	Result Size
$R_1 = Select_{Name='Kinokuniya'}(Merchant)$	2000	1
$R_2 = Join_{ID}(R_1, Events)$	300,001	10,000
$R_{3=}Select_{Category='Books'}(Category)$	175	1
$Join_{Category}(R_2, R_3)$	10,000	8,000
Total	312,176	

Table 2.1: Aggregate Query Row Access Counts

the one shown in Listing 2.1 can generate a huge amount of traffic in a distributed environment. Standard SQL solutions such as database mirroring and replication increase only the database's availability but not its scalability. To deal with the latter problem, the database has to be partitioned horizontally. Horizontally partitioned tables distribute sets of rows across different databases often residing on different servers. This scheme decreases the level of contention between read and write operations at the expense of consistency and increased bandwidth requirement of composite queries such as the one used in the aggregation example. The rest of the chapter will provide an overview of the available approaches to tackle the aggregation problem in the NoSQL world as well as their limitations.

2.2 Data Model

The data model used in Section 2.1 can be formally described via *data cubes*. In online analytical processing, a data cube defines the data model in several dimensions so that the stored information can be manipulated from different perspectives. There are two table types– dimension tables and fact tables. The former type stores non-volatile information while the latter represents time-variant data.

Figure 2.3 shows the data cube describing the data from the example in Section 2.1. As evident from the diagram, there are three dimension tables describing the information in terms of advertised items and their corresponding categories and sellers. Even though the data stored in these tables is not immutable, changes occur very rarely and are triggered by events such as user account creations or modifications. The facts from which the cost is computed are present in the *Events* table which is periodically updated.

It is not difficult to see how data cubes can be used for web-application aggregate queries as measures can be performed at any dimension intersection. Informally, the cube dimensions are presented by the *GROUP BY* statement in SQL and the model can readily be applied for representation of totals such as the total sales from a specific category, web page visits by users from a given country, or the best performing seller for some product. The next few paragraphs will look into several existing ways in which this problem can be tackled.

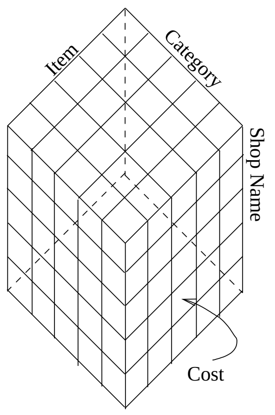


Figure 2.3: Data Cube

2.3 Execution Environment

Unfortunately, computing the aggregate query shown on Listing 2.1 is far from trivial even with a modest amount of traffic such as several hundred views per second. The main reason for the poor performance is the fact that queries involving large tables requiring join and aggregate operations are expensive both in terms of computational cost and communication latency. As noted in Section 2.1, the system's performance can be perceivably improved by scaling horizontally the application's database tier. First, a very large fact table will be distributed across multiple servers, decreasing the workload on any specific instance. Second, the aggregate computation ideally is offloaded to multiple nodes. These two goals cannot be achieved by standard RDBMSs as their heavy-weight transaction handling results in a poor performance and they cannot take advantage of the elasticity achieved by relaxing consistency requirements. The rest of this thesis will investigate data cube aggregate computations for web-applications using a key-value data store such as HBase for their database tier.

HBase is an open-source, distributed, large-scale, key-value data store based on Google's BigTable. The data is organized in labeled tables; each data point is referred to as a cell. As cells are simple byte arrays, the data types widely used by RDBMS are eliminated and the data interpretation is delegated to the programmer. Each table cell is indexed by its unique row key and each row can have an arbitrary number of cells.

Intuitively, an HBase application has to use a data model resembling a distributed, sparse, multidimensional hash table. Each row is indexed with a row key, a column key, and a time-stamp. A table allows maintaining different data versions by storing any number of time-stamped versions of each cell. A cell can be uniquely identified by its corresponding row key and column family and qualifier, and time-stamp. The column keys are in the form

”column-family:qualifier”, where the column-family is one of a number of fixed column-families defined by the table’s schema, and the qualifier is an arbitrary string specified by the application.

It is easy to see why an HBase-like data store is suitable as the storage tier of a web application in the Cloud. New resources are automatically handled by the data store as all operations happen on a per row basis and theoretically, the data store can achieve unlimited horizontal scalability. Nevertheless, aggregate functionality cannot be implemented directly without a framework for breaking-up the queries so that local resources are utilized as much as possible without sacrificing bandwidth.

Chapter 3

State of the Art Data Aggregation for the Cloud

This chapter reviews the MapReduce framework which has been studied extensively and is widely used for large-scale data analysis. Even though, it can efficiently process terabytes of data, the MapReduce technique turns out to be unsuitable for web applications which often have to perform a set of predictable aggregations with low-latency requirements.

3.1 Aggregation with the MapReduce Framework

The MapReduce framework is used for large-scale distributed computation on computer clusters as well as on the Cloud. It can efficiently aggregate terabytes of data and yields in theory unlimited scalability. The model achieves massive parallelization by splitting the input into numerous pieces each of which is assigned to a computational node from the cluster (Figure 3.1). The map function runs on each node and emits key/value pairs. These are passed to the reduce function where the values are collated.

The MapReduce framework can be readily used for computing aggregate queries by using the *Map* and *Reduce* functions to build an execution plan equivalent to the original query. For example, the following SQL query,

SELECT CatId, **SUM**(Cost)**FROM** Events **GROUP BY** Category,
can be computed in two phases as follows:

1. The *Map* phase reads all records from the *Category* table and outputs records consisting of a "key", to which the *CatId* value is assigned, and "value", equal to "1".

2. The *Reduce* phase consumes the key-value pairs generated by the *Map* function and aggregates them so that records with the same key occur together. The *Reduce* phase accumulate all the 1's to compute the final count.

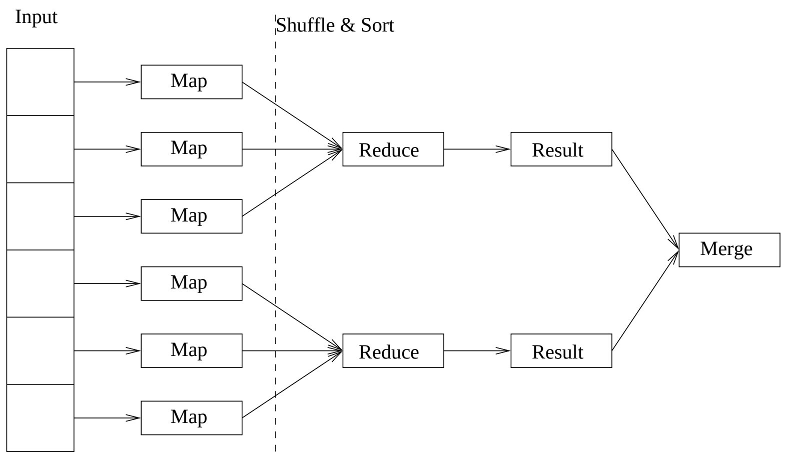


Figure 3.1: MapReduce Framework

Moreover, if the aggregate query is commutative and associative, this approach can be further improved by computing in-network partial aggregtes–intermediate results which are combined to generate the final aggregation [24]. The following algorithm uses "combiner" functions to generate the partial aggregtes within the MapReduce framework and is applicable to *Average*, *Count*, *Maximum*, *Median*, *Minimum*, and *Sum* functions [9]:

1. *Map*: applied on all input files to extract keys and records;
2. *InitialReduce*: operates on a sequence of records of the same type and with the same key to output a partial aggregation;
3. *Combine*: operates on a sequence of partial aggregations to output a combined partial aggregation;
4. *Reduce*: operates on a sequence of partial aggregations to compute the final aggregation result .

Based on the approach outlined above, any aggregate query can be expressed as:

Listing 3.1: Using *MapReduce* to compute an aggregate function

```
SELECT Reduce() AS R FROM (SELECT Map() FROM T) GROUP BY
    Rkey
```

Another advantage of the MapReduce framework over RDBMS is its ability to handle an arbitrary numbers of keys for each record per aggregate operation. Thus, MapReduce can handle in a single pass more complex queries which usually require two passes over the query data by SQL.

The MapReduce Framework for Scalable Web Applications–Challenges While the MapReduce framework theoretically provides unlimited horizontal scalability and fault tolerance, there are several challenges to utilizing the framework for aggregate queries in Cloud-based web-applications. One of the major drawbacks of most MapReduce implementations such as the ones provided in Hadoop and MongoDB [17] is that they operate in batch mode and are suitable only for enterprise analytics computations which often run during the night at data warehouses.

Many web applications need to compute simple low-latency aggregations involving their fact and dimension tables. From the discussion above, it is evident that the proposed aggregate query implementation in Section 3.1 is not a silver bullet for light-weight aggregations. First, not all NoSQL architectures are built upon the MapReduce framework; in these cases, "partial aggregation" can be achieved by exploring standard solutions such as the two-phase aggregation. Second, because the map function needs to inspect all input rows upon each new query, the MapReduce framework is unsuitable for low latency applications. Third, MapReduce needs to perform a complete computation even if a fraction of the input data has changed. And finally and most importantly, to be usable by a large range of web applications, aggregate queries need to provide transaction guarantees–the scalability of the MapReduce framework comes exactly because the framework provides no strong consistency.

3.2 Aggregate Queries in Cloud-based Web Applications

Based on the short example at the beginning of last chapter and subsequent MapReduce discussion, an aggregate query implementation for web applications in the Cloud should have the following properties:

- Strong transaction guarantees and consistency (for instance, updating simultaneously the fact and dimension tables should not lead to billing errors in the Dashboard application);
- Online aggregate query processing. As most web application aggregate queries do not involve heavy transactions, batch processing frameworks such as MapReduce do not provide any advantages;
- Incremental updates. Web application aggregate queries may involve fact tables which by definition are volatile; ideally the aggregate computation should not start from scratch whenever a table used by the computation gets modified.

In the relational database world, the requirements listed above can be achieved by maintaining materialized views–a commonly used technique. As we will see in the next chapter, the theory behind incremental aggregate view maintenance can be readily used in non-relational data stores and it serves as a foundation for this thesis.

Chapter 4

Materialized View Maintenance—A Theoretical Perspective

4.1 Materialized Views

The motivating example from Chapter 2 illustrates a very common database problem consisting of improving the query performance of a stream of repeating aggregate queries over a set of changing tables. Intuitively, the results of queries such as the one shown on Listing 2.1 can be stored and updated only when the *Events*, *Merchant*, *Products*, or *Categories* tables get modified. This approach eliminates the necessity of repeating all the computations and row access operations shown on Table 2.1.

Formally, the technique discussed in the previous paragraph is defined as materialized views which precompute expensive operations prior to their execution and store the results in the database. When the query from Listing 2.1 is executed, the storage engine creates a query plan against the base tables participating in the aggregation. A materialized view takes a different approach in which the query result is cached as a concrete table that is periodically updated. The view update itself can be either entirely re-computed from scratch or maintained incrementally depending on the query properties. The latter approach is significantly cheaper, especially when the updates are smaller than the base tables [6]. Incremental view maintenance for aggregate expressions has been extensively studied by Gupta and Mumick [15]. This chapter provides an overview of aggregate view computation and propagation regardless of the underlying storage engine implementation. Before continuing with the discussion of aggregate views, it is necessary to review some general bag algebra notation relevant both in the relational world and schemaless storage engines as all common database operations can be represented as bag algebra expressions.

4.2 Notation and Terminology

A multiset is a generalization of a set allowing multiple membership. It can be also referred to as a bag. There are several bag algebraic operators and terms which are useful in discussing aggregate view maintenance:

- The **bag union operator**, $\uplus$, is equivalent to the summation of items without duplicate removal. For example, $R_1 \uplus R_2$ represents the concatenation of table rows in a data store;

- The **monus operator** $\dot{-}$ is a variant of subtraction of one bag from another. For example, $B_1 \dot{-} B_2$ evaluates to a B such that for every $d : s$, $count(d,B) = count(d,B_1) - count(d,B_2)$ if $count(d,B_1) > count(d,B_2)$; and $count(d,B) = 0$ otherwise [16]. Intuitively, an element appears in the difference $B_1 \dot{-} B_2$ of two bags as many times as it appears in B_1, minus the number of times it appears in B_2;

- $\triangle\ E$ denotes **insertions** into a bag algebra expression. In a storage engine, the expression evaluates the the rows inserted into the database;

- ∇E denotes **deletions** from a bag algebra expression. In a storage engine, the expression evaluates the the rows deleted from the database;

- $\sigma_p E$ denotes **selection** from E on condition p. Informally, the expression picks rows from the expression E if the condition expression p evaluates to true;

- $\Pi_A E$ denotes the **duplicate preserving projection** on a set of attributes A from E's schema. Intuitively, the operator selects the data from all columns labeled with the attributes enumerated in A;

- $\pi_{a_1 \cdots a_n} E$ denotes the **generalized projection** operator over a set of attributes $a_1 \cdots a_n$. The generalized projection operator restricts all tuples in E to elements matching $a_1 \cdots a_n$. Informally, $\pi_A E$ is algebraically equivalent to a *GROUP BY* statement in SQL [13];

- $E_1 \bowtie_J E_2$ denotes a **join** operation on condition J.

Using the introduced bag algebra notation, the SQL statement from Listing 2.1 on page 9 can be rewritten as $\pi_{sum(cost)}(\sigma_{merchantId='Kinokuniya'}(Events \bowtie_{merchantId} Merchant \bowtie_{catId} Categories))$. In this equation, the $\pi_{sum(cost)}$ expression represents an aggregate function on the *cost* attribute; in other words, the operation describes a generalized projection over a summation function. Intuitively, the sum over a set of *GROUP BY* attributes can be incrementally maintained by memorizing the aggregate result and updating its value whenever a row is inserted or deleted from a base table. For example, whenever a new event is added to the *Events* table, the storage engine can compute any new tuples resulting from $\sigma_{merchantId='Kinokuniya'}(\triangle Events \bowtie_{merchantId} Merchant \bowtie_{catId} Categories)$ and update the summation result. Formally, depending on the memory requirements for re-evaluating an aggregation after applying an update $(\triangle B_x, \nabla B_x)$ to some base relation B, aggregate functions are classified into three categories [11]:

- **Distributive** functions produce results which after an update can be re-computed from the preceding aggregate result;
- **Algebraic** functions yield results which can be computed after an update operation by using a small constant storage;
- **Holistic** functions yield results which cannot be re-computed after an update by using some bounded storage work space.

For example, the SUM and COUNT functions are distributive both for insert and delete operations while MIN and MAX are distributive only for inserts. The latter two functions may need to inspect all the records of the affected groups of a delete operation and are therefore holistic functions with respect to deletions. The evaluation of AVG uses the COUNT function which consumes constant space; therefore, AVG is an algebraic function.

This thesis work elaborates on distributive and algebraic aggregate functions for Cloud-based web applications.

4.3 View Maintenance via Change Tables

Materialized views are a common approach for improving the performance of computationally and data intensive queries in large databases in the petabyte range. As discussed in Section 4.1, an efficient way to update a materialized view after changes in its base relations is to compute only the set of rows to be deleted and inserted from the view. Update operations do not need a special implementation as effectively, an update operation on an existing table row can be transformed into deletion followed by insertion: $R_{A'_1,A'_2..A'_n} \rightarrow R_{A"_1,A"_2..A"_n} \equiv \nabla R_{A'_1,A'_2..A'_n} \uplus \triangle R_{A"_1,A"_2..A"_n}$. In [15] Gupta and Mumick compute the views of distributive aggregate functions by using incremental view maintenance via change tables which are applied to the relevant views using special refresh operators. The following sections outline this approach.

4.3.1 Change Tables

A change transaction t can defined by the expression $R_i \leftarrow (R_i \dot{-} \nabla R_i) \uplus \triangle R_i$ where $R = \{R_1..R_n\}$ is the set of relations defining a database. In other words, t contains the set of insertions and deletions in a database update operation. Let V be a bag algebra expression over R and $New(V,t) = (V \dot{-} \triangledown(V,t)) \uplus \triangle(V,t)$ be the refresh expression used to compute the new value of V after a change transaction t [12]. Intuitively, the last expression can be interpreted in the following way: a change transaction t modifies a view V by removing $\triangledown V$ from V (monus $\dot{-}$ operation) and inserting $\triangle V$. As each transaction can be rewritten as a collection of insertions and deletions from the base relations, a natural way to capture the $\triangle$ and $\triangledown$ operations is by using a change table $\Box(V,t)$ representing the difference between V and $New(V,t)$.

Gupta and Mumick refine the expression for $New(V,t)$ by introducing the *refresh* operator $\sqcup_\theta^U$ such that $New(V,t) = V \sqcup_\theta^U \Box(V,t)$. The refresh operator matches rows from the change table $\Box(V,t)$ with the original view V according to the join conditions specified in θ and applies update functions U.

4.3.2 Refresh Operator

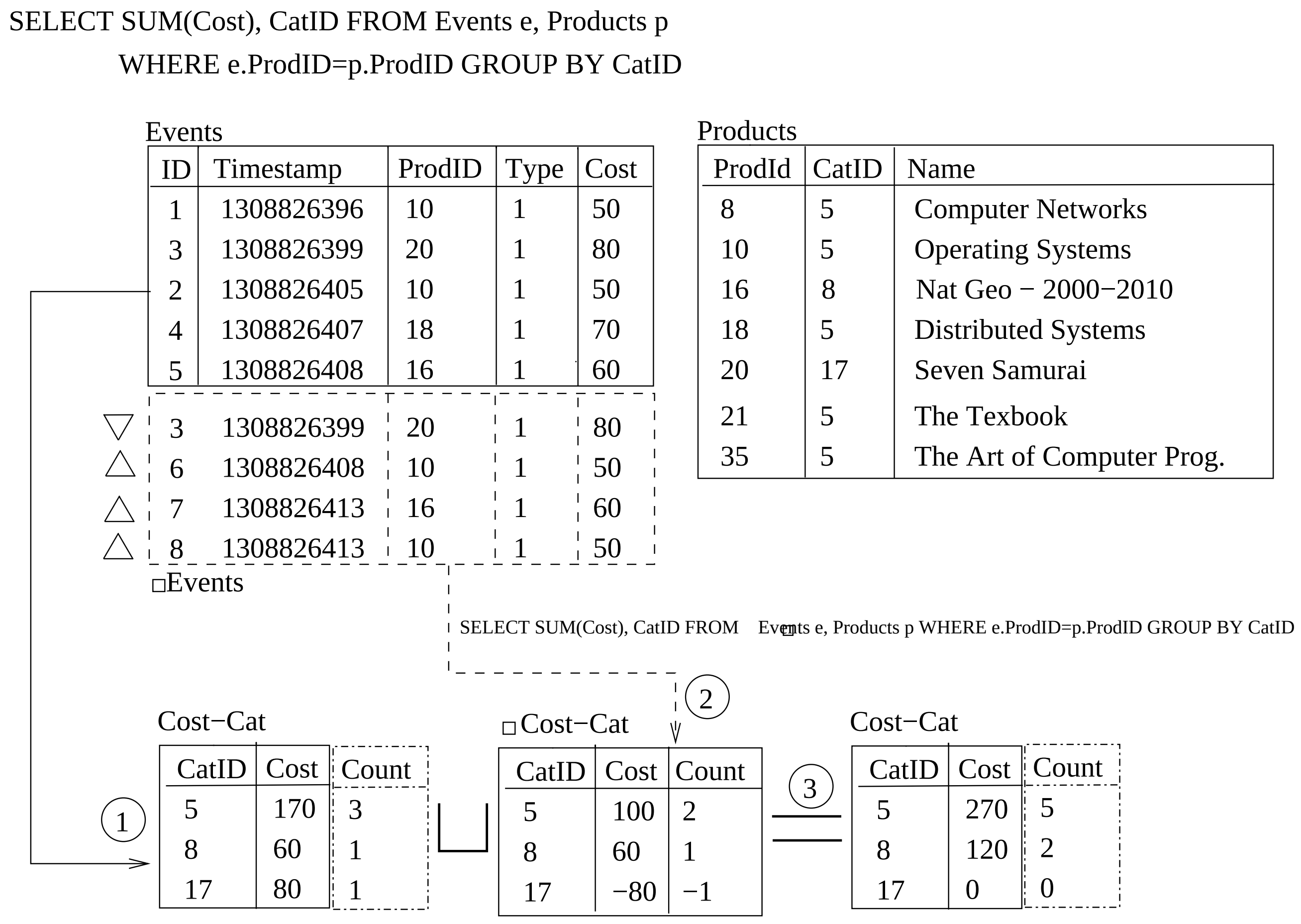
The refresh operator is defined by a pair of matching conditions θ and update function U. More specifically θ is a pair of join conditions $\mathscr{J}_1$ and $\mathscr{J}_2$ such that tuples generated by $V \bowtie_{\mathscr{J}_1} \Box V$ are changed in V according to the update specification U while matches generated by $V \bowtie_{\mathscr{J}_2} \Box V$ are deleted. Any unmatched tuples from the change table $\Box V$ are inserted into V. The update function U is defined as the collection $U = \{(A_{i_1}, f_1), (A_{i_2}, f_2)..(A_{i_k}, f_k)\}$ where $A_{i_1}..A_{i_k}$ are the attributes of V and $f_1..f_k$ are binary functions. During the refresh operation, each attribute of V $A_{i_j}$is changed to $f_j(v(A_{i_j}), \Box v(A_{i_j}))$, where $v(A_{i_j})$ and $\Box v(A_{i_j})$ denote the values of attribute A_{i_j} in V and $\Box V$ respectively.

4.3.3 Change Table Computation

As previously discussed, whenever a base table receives an update consisting of insertions and deletions, the result of an aggregate query involving the modified table needs to be reevaluated. Formally, an aggregation over a Select-Project-Join query can be expressed as $\pi_{G,f(AggAttr \in A)}(\Pi_A(\sigma_p(B_1 \bowtie_J B_2)))$, where G and A correspond to the *GROUP BY* and projection attributes respectively. If R substitutes the result set produced by the SPJ relation $\Pi_A(\sigma_p(B_1 \bowtie_J B_2))$, the aggregate change table capturing the "delta" between the old and new results is

$$\Box V = \pi_{G,f(AggAttr \in A),sum(\_count)}(\Pi_{G,A,\_count=1}(\triangle R) \uplus \Pi_{G,\bar{A},\_count=-1}(\triangledown R)) \tag{4.1}$$

The expression $\Pi_{G,A,\_count=1}(\triangle R) \uplus \Pi_{G,\bar{A},\_count=-1}(\triangledown R)$ from Equation 4.1 selects all *GROUP BY* and aggregation attribute values from the set of inserted and deleted rows from the SPJ result set R. In case of row deletion ($\triangledown R$), the projected attribute values have been negated so that the deletion is accounted for by the generalized projection π.


SELECT SUM(Cost), CatID FROM Events e, Products p
WHERE e.ProdID=p.ProdID GROUP BY CatID

Events

ID	Timestamp	ProdID	Type	Cost
1	1308826396	10	1	50
3	1308826399	20	1	80
2	1308826405	10	1	50
4	1308826407	18	1	70
5	1308826408	16	1	60

□Events

▽	3	1308826399	20	1	80
△	6	1308826408	10	1	50
△	7	1308826413	16	1	60
△	8	1308826413	10	1	50

Products

ProdId	CatID	Name
8	5	Computer Networks
10	5	Operating Systems
16	8	Nat Geo − 2000−2010
18	5	Distributed Systems
20	17	Seven Samurai
21	5	The Texbook
35	5	The Art of Computer Prog.

SELECT SUM(Cost), CatID FROM Ev□nts e, Products p WHERE e.ProdID=p.ProdID GROUP BY CatID

Cost−Cat

CatID	Cost	Count
5	170	3
8	60	1
17	80	1

□Cost−Cat

CatID	Cost	Count
5	100	2
8	60	1
17	−80	−1

Cost−Cat

CatID	Cost	Count
5	270	5
8	120	2
17	0	0

Figure 4.1: Change Table Derivation after a Base Table Update

What remains to be computed are the rows to be inserted and deleted from the result of the SPJ query. Effectively, the database engine can apply the selection and join operators on a projection of the inserted and deleted rows from the base relation. The last follows from equation 4.2:

$$\sigma(B_1^{new} \bowtie B_2) = \sigma(((B_1^{old} \dot{-} \nabla B_1) \uplus \triangle B_1) \bowtie B_2) \Longleftrightarrow$$
$$\sigma(B_1^{new} \bowtie B_2) = (\sigma(B_1^{new} \bowtie B_2) \dot{-} \sigma(\nabla B_1 \bowtie B_2)) \uplus \sigma(\triangle B_1 \bowtie B_2) \Longrightarrow$$
$$\begin{cases} \triangle\, \sigma(B_1^{old} \bowtie B_2) = \triangle R = \sigma(\triangle B_1 \bowtie B_2) \\ \nabla \sigma(B_1^{old} \bowtie B_2) = \nabla R = \sigma(\nabla B_1 \bowtie B_2) \end{cases} \tag{4.2}$$

Figure 4.1 illustrates the usage of the expressions provided in Equations 4.2 and 4.1. The example evaluates “**SELECT SUM**(Cost),CatID **FROM** Events e, Products p **WHERE** e.ProdID=p.ProdID **GROUP BY** CatID” which is the total cost per category, stored in the *Cost-Cat* table, from the *Events* fact table and *Products* dimension table. The aggregation is performed in three phases discussed in the following paragraphs.

Initially, the *Cost-Cat* table has no entries. To compute the aggregation, the database engine has to inspect the *Events* and *Products* tables from scratch as shown in ① on Figure 4.1. In addition to the attributes specified in the original query, the aggregate result table needs to maintain a counter for the SPJ rows grouped by the *GROUP BY* attributes. For example, $\Pi_{ProdID,Cost,CatID}(Events \bowtie_{ProdID} Products)$ generates 3 results from category 5, 1 result from category 8, and 1 result from category 17.

After the initial aggregation has been computed, the database can readily maintain the aggregate result via change tables (② on Figure 4.1). The second phase computes the change table $\Box$*Cost-Cat* from the *Events* insertions and deletions, prefixed with $\triangle$ and $\triangledown$ in the diagram. In the example shown on the diagram, there have been 3 insertions and one deletion. The change table is computed by joining the insertions and deletions with *Products* and aggregating the result. The aggregation attributes and counts resulting from deletions have to be negated.

After the database engine has computed the aggregate result change table, it is ready to apply it to the result itself (② on Figure 4.1). In order to complete the update, the database engine evaluates $\Box Cost\text{-}Cat \bowtie_{ProdID} Cost\text{-}Cat$. Any change table rows which are not present in the join result set have to be inserted into the aggregate view table. Otherwise, the matching attributes from $\Box$*Cost-Cat* and *Cost-Cat* are simply summed and the corresponding aggregate view table row is updated or deleted if the *count* attribute is *0*.

4.3.4 Change Table Propagation

Finally, change tables such as the one generated in the previous section can be readily propagated through relational operators using the equations listed on Table 4.1 [15]. The major relational operators used in data stores are *Selection*, *Projection*, *Inner Join*, *Bag Union*, and *Aggregation*. Once a change table for an aggregation has been computed, it can be propagated and applied without reevaluating the whole aggregate expression.

Table 4.1: Change Propagation Equations [15]

Type	V	V_{new}	Refresh	$\Box V$
Selection	$\sigma_p(E_1)$	$\sigma_p(E_1 \sqcup^U_\Theta \Box E_{1)}$	$V \sqcup^U_\theta \sigma_p(\Box E_1)$	$\sigma_p(\Box E_1)$
Projection	$\Pi_A(E_1)$	$\Pi_A(E_1 \sqcup^U_\Theta \Box E_1)$	$V \sqcup^U_\theta \Pi_A(\Box E_1)$	$\Pi_A(\Box E_1)$
Inner Join	$E_1 \bowtie_J E_2$	$(E_1 \sqcup^U_\Theta \Box E_1) \bowtie_J E_2$	$V \sqcup^U_{\theta_1} (\Box E_1 \bowtie_J E_2)$	$\Box E_1 \bowtie_J E_2$
Bag Union	$E_1 \uplus E_2$	$(E_1 \sqcup^U_\Theta \Box E_1) \uplus E_2$	$((V - E_2) \sqcup^U_\theta \Box E_1) \uplus E_2$	$\Box E_1$
Aggregation	$\pi_{G',F}(E_1)$	$\pi_{G',F}(E_1 \sqcup^U_\Theta \Box E_1)$	$V \sqcup^{U_3}_{\theta_3} \pi_{G',F}(\Box E_1)$	$\pi_{G',F}(\Box E_1)$

Chapter 5

System Design and API

This chapter discusses the implementation of the aggregate view maintenance techniques from Chapter 4. The main algorithm for automatically keeping aggregate views can be implemented either synchronously or asynchronously. The main difference between the two approaches is that in the former case, the base table update and change table $\Box V$ computation and application, $E_1 \sqcup^{U}_{\Theta} \Box E_1$, are carried out in the same transaction while in the latter case, $\Box V$ and $E_1 \sqcup^{U}_{\Theta} \Box E_1$ are deferred to a separate asynchronous transaction. Nevertheless, in both cases the Cloud data store and its middleware need to support consistent read/write transactions.

Currently, CloudTPS is the only middleware providing strongly consistent transactions for web applications using non-relational data stores such as HBase and SimpleDB. The rest of this chapter provides details about the implementation of the synchronous and asynchronous view maintenance algorithms on CloudTPS.

5.1 Overview of CloudTPS

CloudTPS is a middleware for key-value data stores which provides strongly consistent and fault-tolerant transactions for web applications. The read-only transactions support look-ups by primary and secondary keys as well as equi-joins while the read/write transactions support insertion, deletion, and update transactions on individual rows. The read and write transactions are received by read/write and read-only transaction managers which distribute the work over a set of local transaction manager (LTM) nodes (Figure 5.1). Each LTM is responsible for handling a set of table rows identifiable by their primary keys. Both read and write transactions are submitted to the LTMs using the two phase commit protocol with the transaction manager serving as a coordinator. To ensure strong consistency, each transaction receives a global timestamp which is used for creating an execution order on the

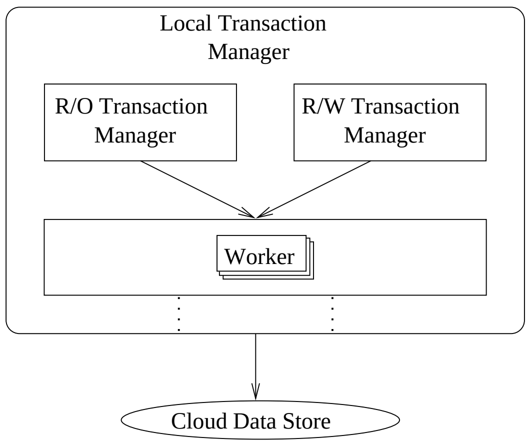


Figure 5.1: CloudTPS Architecture

LTMs: transactions with newer timestamps than the timestamp of the current transaction are queued in FIFO order while all transactions with “old” timestamps are aborted.

To implement the synchronous and asynchronous change table view maintenance algorithms, the CloudTPS architecture needs to support more complex read/write operations consisting of transactions which execute an SPJ query plan and use the result as the input of update, insert, or delete sub-transactions. This modification is necessary in order to enforce consistency between the base relation updates and the maintained aggregate views. The modified read/write transaction mechanism is similar to the existing approach towards index table maintenance in CloudTPS and requires the implementation of a new transaction manager which uses the two phase commit protocol to coordinate the read and write sub-transactions. The next section discusses in detail the proposed CloudTPS modification.

5.2 Aggregate Queries for CloudTPS

The basic change table mechanism for CloudTPS is similar to the one provided by Gupta et al and discussed in Chapter 4. For a given aggregate query $\pi_{func(AggAttr),GB}(\sigma_{SelAtt}(B_1 \bowtie_{JoinCond_1} B_2 \bowtie_{JoinCond_2} B_3 \cdots))$, CloudTPS needs to monitor the base relations $B_1 \ldots B_n$ for changes and re-evaluate the aggregate view upon update. Algorithm 5.1 describes the synchronous version of this approach. *Step 1* is executed once during the system’s initialization and it computes from scratch all aggregate views. After the initial step, all views can be incrementally updated from the old view and update values. *Step 2* as shown on the diagram is executed as a single transaction and thus the performance of update queries is strongly dependent on the complexity of the materialized views in which the updated base relations participate: the heavier the aggregate expression is, the more intensive *step 2.a* is.

Algorithm 5.1 Basic Aggregate View Maintenance in CloudTPS .

```
1. Startup: Compute $V$ as $\pi_{func(AggAttr),GB}(\sigma_{SelAtt}(B_1 \bowtie_{JoinCond_1} B_2 \bowtie_{JoinCond_2} B_3 \cdots))$
2. Coordinator LTM: Upon a base relation, $B_k$, update
    (a) Compute $\Box B_k$, defined as $\triangle B_k \uplus \nabla B_k$
    (b) Compute the aggregate change table, $\Box V$ as $\pi_{func(AggAttr),GB,count}(\sigma_{SelAtt}(B_1 \bowtie_{JoinCond_1} B_2 \bowtie_{JoinCond_2} B_3 \cdots \Box B_k \bowtie_{JoinCond_k} ...))$
    (c) Apply $\Box V$ to V using the special join operator
```

The write performance upon base table updates can be generally improved by modifying the synchronous algorithm into Algorithm 5.2. The main difference between algorithms 5.1 and 5.2 is that the asynchronous version computes the aggregate view change table asynchronously and maintains additional CloudTPS tables storing $\Box B_1, \Box B_2 \cdots, \Box B_n$. The system initialization computes all aggregate views from scratch and clears the base relation change tables. *Step 2* is executed as a transaction whenever a base table is updated and consists of updating the corresponding base relation change table $\Box B_k$ and B_k itself. The actual work performed by CloudTPS during write operations is significantly less than the workload in the synchronous algorithm as the change table computation which performs several join operations is postponed. The actual computation of $\Box V$ and $\Box V$'s application to the aggregate view table is performed in the last phase. In *Step 3*, the algorithm computes $\Box V$ from the base table change tables and the current value of the base tables. The asynchronous phase of the algorithm introduces a relatively heavy workload of *O(n)* join query plans.

5.3 Implementation

The CloudTPS architecture supports equi-join read-only queries and simple read/write transactions consisting of sub-transactions modifying at most one row. The two operations are handled by a read-only and read/write transaction manager respectively. Even though this approach offers greater efficiency when handling the two query types, there is no mechanism allowing transactional execution of update operations using the input of read-only query plans. This problem can be addressed by either introducing table locking preempting concurrent same-table updates or a new transaction manager capable of handling the new read/write query. Unfortunately, despite its simplicity, the former approach is prohibitively expensive as it sacrifices the read/write horizontal scalability by

Algorithm 5.2 Asynchronous Aggregate View Maintenance in CloudTPS

```
1. Startup:
    (a) Compute $V$ as $\pi_{func(AggAttr),GB}(\sigma_{SelAtt}(B_1 \bowtie_{JoinCond_1} B_2 \bowtie_{JoinCond_2} B_3 \cdots))$
    (b) Clear $\Box B_1, \Box B_2 \cdots, \Box B_n$
2. Coordinator LTM: Upon a base relation,$B_k$, update
    (a) Compute $\Box B_k$, defined as $\triangle B_k \uplus \triangledown B_k$
    (b) Commit $\Box B_k$ as an ordinary CloudTPS table
3. Coordinator LTM: Asynchronous
    (a) Retrieve base relations change tables $\Box B_1, \Box B_2 \cdots, \Box B_n$
    (b) Compute $\Box V = \pi_{func(AggAttr),GB}(\sigma_{SelAtt}(\Box B_1 \bowtie_{JoinCond_1} B_2 \uplus B_1 \bowtie_{JoinCond_1} \Box B_2 . \dot{-} \Box B_1 \bowtie_{JoinCond_1} \Box B_2)$
    (c) Clear $\Box B_1, \Box B_2 \cdots, \Box B_n$
    (d) Apply $\Box V$ to V using the special join operator
```

decreasing the update transaction granularity. Thus the only viable solution is employing a mechanism capable of executing update transactions consuming the output of read-only operations. The rest of this section discusses the implementation of the new query manager handling transactions similar to the ones described in *steps 2* and *3* of algorithms 5.1 and 5.2.

5.3.1 Synchronous Computation

The synchronous version of the aggregate view maintenance algorithm maintains only one CloudTPS table representing the materialized aggregate view V. Both the view and base table change tables $\Box V$ and $\Box B_1 \ldots \Box B_n$ are maintained in memory only during the update operations and do not require creating special CloudTPS tables. As the size of the change tables is relatively small due to the small-sized transactions characteristic for most web applications, these tables can be maintained on a single node serving as a transaction coordinator. Similarly to the read-only and read/write transaction managers, the aggregate view maintenance transaction manager uses a modified version of the two-phase commit protocol in order to guarantee consistency. Thus, when a node crashes, the only penalty incurred is restarting the computation. During the first phase of the protocol's execution, the coordinator node submits updates to the LTM participants. However, instead of sending only *ACK* and *NACK* messages, the LTMs return additional information resulting in adding more sub-transactions to the running transaction. The additional sub-transactions implement the logic in the view maintenance algorithm. If all the LTMs, including the ones which have been added later during the transaction's course, send acknowledgments to the coordinator, the transaction is ready to be committed. The second phase commits all base table updates and the corresponding aggregate view table modifications.

The modified two phase commit protocol is shown of Figure 5.2. When a client-side application submits a read/write transaction modifying table A and C (not shown), the transaction manager executes the two phase commit protocol. The data item location and the transaction management mechanism is similar to the one present in the unmodified CloudTPS transaction manager:

1. In order to ensure consistency, each transaction is assigned a global timestamp which is used for queuing conflicting transactions in FIFO order. Transactions with "stale" timestamps are aborted;

2. The coordinator identifies all LTMs responsible for each sub-transaction and builds all relevant sub-transaction groups to be submitted;

3. Each LTM votes whether its sub-transactions can be committed.

However, instead of simply committing the base table updates upon consensus among the LTMs, the coordinator needs to compute sequentially the base table's change tables, the aggregate view change table ($\Box V$), and the updated view after the application of $\Box V$. As the change tables can easily fit into a single node's main memory, the coordinator proceeds with the join algorithm used in the read-only transaction manager to compute $\sigma_{Attributes} \Box B_1 \bowtie B_2 \ldots B_n$. Finally, after $\Box V = \pi_{Agg,Func,\_count} \sigma_{Attributes} \Box B_1 \bowtie B_2 \ldots B_n$ has been computed, the algorithm proceeds by adding to the current transaction more read-write sub-transaction in order to apply $\Box V$. Upon failure, the coordinator aborts the transaction and consequently, the CloudTPS tables remain consistent as neither the base table modifications nor the aggregate view is committed.

The approach described in the previous paragraphs is shown on Figure 5.2. As shown on the diagram, A and B are base tables which participate in the join query $\sigma_{ID,Cost,Prod,Cat} A \bowtie_{Prod} B$ defined by some client application. At a certain point during its execution, the web-application submits to LTM_3 a read/write transactions modifying tables A and C (not shown). LTM_3 serves as the transaction's coordinator and needs to compute $\Box A$, $\Box V$, and apply the change table to V. The transaction is executed as follows:

1. LTM_3 starts the two-phase commit protocol and submits the base table updates to the responsible LTMs (LTM_1,LTM_2,LTM_4) ①. For simplicity, the CloudTPS secondary key index maintenance mechanism has been omitted from the diagram;

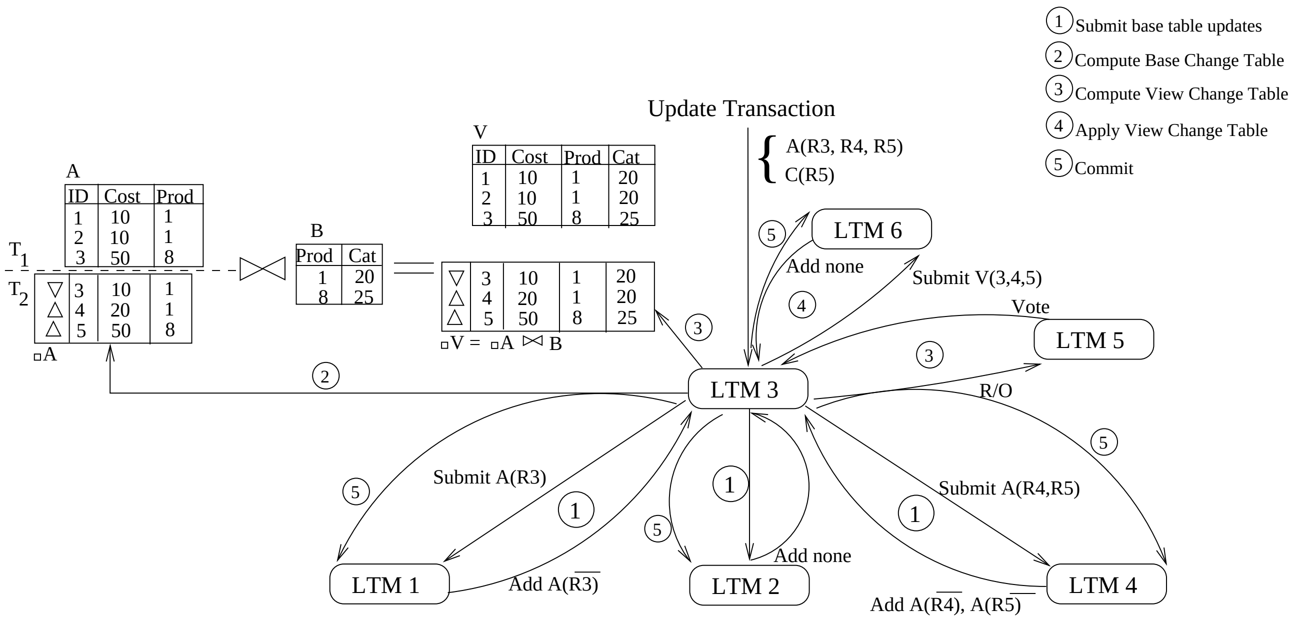


Figure 5.2: Synchronous Computation Two-Phase Commit Protocol

2. LTM_1,LTM_2, and LTM_4 piggyback on their ACK messages all the attributes associated with the base table being updated so that the coordinator can compute $\Box A$ ②. This step is necessary because a read/write transaction should not necessarily contain all the attributes of the rows to be updated. After all votes have been received, the coordinator has the values of all attributes of rows 3,4, and 5 and can proceed to the next step;

3. The coordinator, LTM_3, submits read-only sub-transactions to LTM_5 to compute $\sigma_{ID,Cost,Prod,Cat}\Box A \bowtie_{Prod} B$ ③. For simplicity, the CloudTPS secondary key queries have been omitted from the diagram. The query mechanism is the same as the one employed by the read-only transaction manager. The coordinator submits the query to the LTMs responsible for the data items and if necessary new read-only look-ups are added if secondary keys are used. After all votes have been received, the coordinator computes $\Box V = \pi_{Cost,\_count}\sigma_{ID,Cost,Prod,Cat}\Box A \bowtie_{Prod} B$ and is ready to submit the final read/write sub-transactions in order to update the aggregate view table;

4. The aggregate view maintenance operator $\sqcup^{U}_{\Theta}$ is implemented as a special write operation similar to insertion, deletion, and update and will be described in detail in Section *5.3.2*. After $\Box V$ has been computed in the previous step, the coordinator is ready to update the aggregate view as $\Box V \sqcup^{U}_{\Theta} V$ ④. In the specific example, LTM_3 submits a sub-transaction to LTM_5 as LTM_5 maintains the whole aggregate table;

5. After all involved LTMs have submitted their votes to the coordinator LTM_3, the transaction can be either committed or aborted.

5.3.2 CloudTPS Aggregate View Refresh Operator

As discussed in Section 4.3.2, the aggregate view refresh operator applies the aggregate view change table $\Box V$ to the aggregate view V in order to reflect any base table changes. To this end, both the aggregate view and its change table need to maintain an additional attribute, *_count*, which keeps track of the number of rows aggregated by the *GROUP BY* statement. When applying the change table to the view itself, the refresh operator applies the specified aggregate function with positive change table aggregate attributes for insertions and negative values for deletions. If $\_count = 0$ for some row in the aggregate view, the row has to be deleted. Otherwise, the row needs to be either updated if it previously existed in the view or inserted.

Matching rows from the aggregate view V with rows from its change table $\Box V$ is equivalent to an equi-join operation on the *GROUP BY* attributes. Moreover, the operation can be sped-up by introducing a primary key consisting of the concatenation of the *GROUP BY* attributes so that any secondary key look-ups and index table are eliminated. Thus, V and the corresponding $\Box V$ need to have two additional attributes used internally by CloudTPS: *_groupBy* (primary key) and *_count* (count of aggregated rows). The refresh operator can be readily implemented using the two-phase commit protocol as shown in Algorithm 5.3.

5.3.3 Asynchronous Computation

As discussed in the beginning of this chapter, despite being strongly consistent, the synchronous aggregate view maintenance approach introduces relatively heavy workloads and can become expensive with larger read/write transactions. The main reason behind this performance issue is the large number of sub-transactions the coordinator needs to add in order to complete the a read/write operation on an aggregate base table. The asynchronous version of the computation shown in Algorithm 5.2 offers greater flexibility at the expense of allowing staleness in the aggregate views.

In the asynchronous version of the aggregation computation, CloudTPS needs to maintain base table change tables which are used for the deferred derivation of the aggregate view. Just like in the synchronous algorithm, the coordinator node identifies the LTMs responsible for the data items to be updated and waits for receiving the complete rows after submitting the sub-transactions. However, after receiving the LTM votes, instead of proceeding with computing $\Box V$, the coordinator adds sub-transactions updating the base table change tables and waits for the final votes before committing. This approach decreases considerably the number of sub-transactions submitted by the coordinator node and allows for an asynchronous view update as it preserves enough information to compute $\Box V$.

Algorithm 5.3 Aggregate View Table Refresh

1. The coordinator generates and submits a *refresh* sub-transaction from each $\Box V$ row and uses the primary key to identify the LTM handling potential view table matches.
2. Each LTM performs one of the following:
   (a) Inserts a row if there is no match on the *_groupBy* attribute
   (b) Carries out the aggregate operations if there is a match on the *_groupB*y attribute and does one of the following:
      i. Deletes the row if $\_count^{new} = 0$
      ii. Performs an update if $\_count^{new} \neq 0$
3. Upon LTM consensus the transaction is committed

The second phase of the asynchronous algorithm obtains and applies $\Box V$ using $B_1 \ldots B_n$. As in the synchronous algorithm, a random node is selected for a transaction coordinator which computes $\Box V = \pi_{func(AggAttr),GB}(\sigma_{SelAtt}(\Box B_1 \bowtie_{JoinCond_1} B_2 \uplus B_1 \bowtie_{JoinCond_1} \Box B_2 . \dot{-} \Box B_1 \bowtie_{JoinCond_1} \Box B_2)$. The previous expression has a simple query plan which can be readily handled by the logic of the read-only transaction manager. After the computation, tables $B_1 \ldots B_n$ are cleared and the aggregate view change table is applied as described in Section 5.3.2.

5.4 Aggregate View Maintenance API

The aggregate view maintenance programming interface closely resembles the CloudTPS read-only query plan API. All aggregate queries need to be statically registered in advance so that the initial aggregate views can be computed during the system's initialization. An aggregate query can be defined as a collection of "JoinTable" and "JoinEdge" objects which have the same semantics as defined by Zhou at al. The only difference between declaring a standard select-project-join query plan and aggregate query plan is the additional definition of *GROUP BY* columns and aggregate functions.

Chapter 6

Evaluation

This section provides an experimental evaluation of the synchronous and asynchronous aggregate view maintenance algorithms presented in Chapter 5. The modified CloudTPS architecture was executed on top of HBase running on the DAS-3 computer cluster which comprises 85 nodes running Linux on dual core 2.4 GHz CPUs and using 4 GB memory. Each node used in the experiments functioned either as an LTM or load generator. The LTM role is described in detail in Chapter 5 while the load generator submits update transactions introducing controlled workloads to the CloudTPS framework.

The aggregate view maintenance aggregate algorithm was evaluated both in term of micro-benchmarks covering specific implementation details and the standard TPC-W benchmark providing a typical workload for an e-commerce web application.

6.1 Experimental Setup

Figure 6.1 shows the experimental setup for the aggregate view maintenance synchronous and asynchronous algorithms. For each evaluation run, a subset of the DAS-3 nodes was assigned either a *load generator* or *local transaction manager* role. Each load generator is a single-threaded application generating update transactions modifying a specified number of data items in CloudTPS-managed tables in the Cloud data store.

None of the load generators submits read-only query plans as evaluating an aggregate query consists of a simple look-up in the aggregate view table. Thus both the micro-benchmarks and TPC-W application simply populate their CloudTPS tables and subsequently send updates. As the LTMs are interconnected via a Gigabit LAN, any network latency effects can be ignored. As the view maintenance table update implementation is based on the CloudTPS secondary key index maintenance, based on the two phase commit protocol, fault tolerance and network partitioning has been largely untested as these topics are covered in detail in [23].

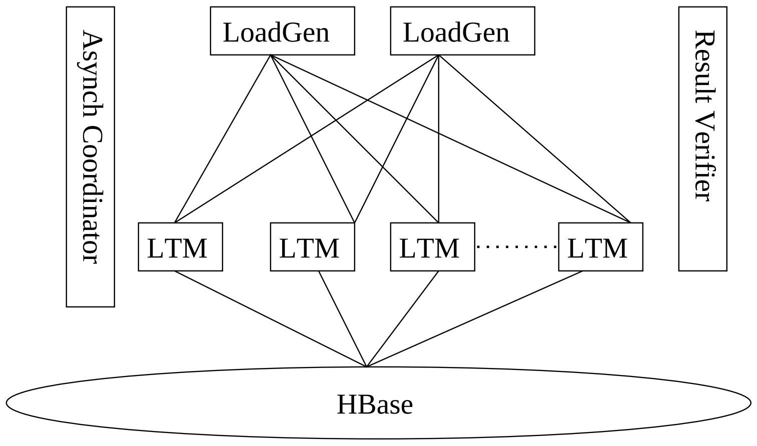


Figure 6.1: Aggregate View Maintenance Experimental Setup

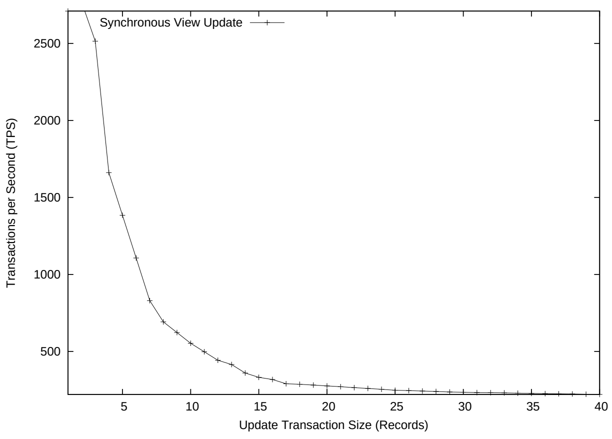
Finally, the asynchronous computation of the asynchronous view maintenance algorithm is driven by a centralized coordinator. In this setup, the coordinator node simply sends view refresh requests to randomly chosen LTMs which serve as the actual transaction coordinators. For simplicity, the current implementation does not provide for any failures at the asynchronous coordinator node. However, a new coordinator can be easily chosen after a crash by using one of the numerous leader election protocols described in the literature [2].

6.2 Correctness

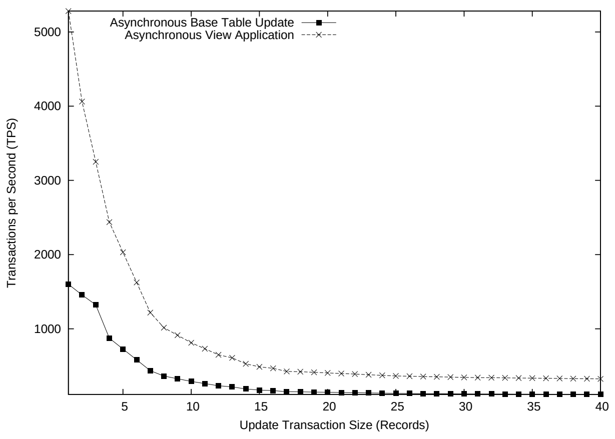
The correctness of the implementation of the synchronous and asynchronous aggregate view maintenance algorithms is done by the *Result Verifier* process which periodically inspects the view tables for consistency issues. To this end, the load generators submit base table updates according to several simple rules which result in an aggregate view table with easily verifiable properties:

- Each load generator submits in arbitrary order n insertion and m deletion transactions with a predetermined numerical value v for some attribute A. At the end of a successful test run, the aggregate view table corresponding to the summation over A will yield a result equal to $count_{LTM} \times (n-m) \times v$;

- Each load generator submits numerical values divisible by some natural number N; consequently, all intermediate results are divisible by N regardless of the transaction interleaving.



(a) Synchronous Algorithm



(b) Asynchronous Algorithm

Figure 6.2: Performance Evaluation of the Synchronous and Asynchronous View Maintenance Algorithms (transactions per second vs update size)

The approach outlined in the previous paragraph is relatively light-weigh and guarantees the absence of false negative results and is effective for discovering the presence of inconsistencies in the aggregate view tables.

6.3 Micro-benchmarks

The aggregate view maintenance algorithm was tested with a simple micro-benchmark modeling the "Dashboard Application" presented in Section 2.1. Because of the relative simplicity of the database schema (Figure 2.1), the load generators have complete control over the complexity of the submitted update transactions. In this setup, the performance of the view maintenance algorithms is entirely determined by the size of the read/write transactions and underlying base tables. The load generator modifies the *Events* table while the *Categories*, *Products*, *Shops*, and *Address* tables remain largely static. The micro-benchmark was setup with *50* shops, 5000 products, 20 categories, and *5,000* events. Each workload generator waits for approximately *1000 ms* before sending the next transaction; CloudTPS was deployed on *20* DAS-3 nodes serving as LTMs.

Figure 6.2 shows the performance of the synchronous and asynchronous algorithms under the conditions described in the previous paragraph. The asynchronous view application workload was measured by generating the base table change tables and applying them until the *100 ms* response time threshold has been exceeded. Intuitively, the synchronous approach comprises considerably more sub-transactions than the asynchronous one as the aggregate view table is updated in the same transaction as the underlying base tables. Thus

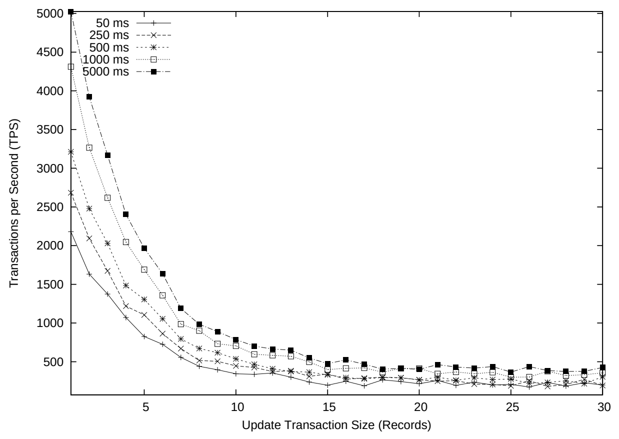


Figure 6.3: Asynchronous View Maintenance with Different Refresh Frequencies (transactions per second vs update size)

the throughput with the synchronous algorithm is considerably lower than the throughput with the algorithm's asynchronous counterpart. On the other hand, the asynchronous algorithm incurs a significant "penalty" which has to be paid during the algorithm's second phase–the actual view update. The last is evident from the curve representing the asynchronous view update: when the change table needs to be applied, the coordinator does much more work than the synchronous coordinator as it has to fetch the base tables change tables first and execute heavier join query plans later.

Despite the heavy workload introduced by the second phase of the asynchronous algorithm, the asynchronous view maintenance approach can be readily applied in applications tolerating eventually consistent aggregate views and exhibiting query locality. Figure 6.3 shows the throughput of the asynchronous algorithm under different view refresh frequencies when a set of *150* base table rows is being updated. The *50 ms* curve is consistent with the preceding discussion as the refresh interval is small enough for the algorithm's performance to degrade to to the synchronous case. However, the *500 ms* graph shows considerable speed-up. The performance of the asynchronous algorithm with a *5000 ms* refresh interval is close to the ideal write performance.

The achievable throughput for the asynchronous algorithm is strongly dependent on the type of the read-write transactions submitted to CloudTPS. Update transactions modifying a small subset of the present rows benefit from large view refresh intervals as the update operations are aggregated in the base table change tables (Figure 6.3). On the other hand, read-write transactions inserting new rows do not see improved performance with decreasing the view refresh frequency as the number of rows present in the base table change tables increase linearly with the count of submitted transactions.

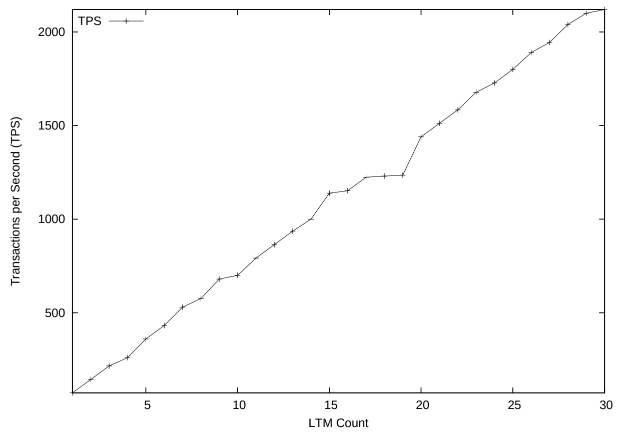


Figure 6.4: Performance Evaluation of the Synchronous View Maintenance Algorithm with TPC-W (transactions per second vs update size)

6.4 Macro-Benchmarks

Section 6.3 discussed the aggregate view maintenance algorithms in the context of a simple micro-benchmark—an approach allowing covering specific functionality such as synchronous and asynchronous updates with specified transaction sizes. Unfortunately, the *Dashboard* application cannot provide a typical web application workload. Instead, the overall system scalability needs to be investigated in terms of the workload provided by a typical e-commerce application such as the TPC-W benchmark.

TPC-W is a transactional web-application simulating an online book seller such as Amazon.com. The benchmark is used for examining the total throughput of a system in transactions per second and stresses all the components of a typical web application–web servers, network, and the storage tier. To test the performance of the aggregate view maintenance algorithm, CloudTPS needs to register aggregate queries such as the total cost of items in a shopping cart or ordered. The actual workload consists only of read-write transactions modifying the base tables. To this end, upon startup the TPC-W tables are populated with *12,000* item records and *100,000* customer records.

Figure 6.4 provides the CloudTPS throughput against the LTM node count when the synchronous algorithm is used. As the diagram shows, adding LTMs increases linearly the throughput. The primary reason for the last is the linear scalability of the underlying read-only queries and read-write transactions. Increasing the LTM count results in spreading the aggregate computation over more nodes. As the transaction cost is negligible due to the high speed LAN, the query response time depends only on the number of accessed records and the record distribution across the LTMs.

Chapter 7

Conclusion

This thesis presented two aggregate view maintenance algorithms for Cloud-based web applications. The main idea behind the algorithms is based on Gupta's work on incremental aggregate view maintenance in relational databases [15]. However, instead of using a centralized solution for carrying out the view maintenance, we demonstrated the feasibility of distributing the workload over a set of local transaction managers. To this end, the algorithm maintains several data structures and computes and applies aggregate change tables via the two-phase commit protocol. This implementation approach has been derived from the CloudTPS join and update operations which have been shown to be strongly consistent and scale horizontally for small-sized transactions which are characteristic for web applications.

As shown in Chapter 6, both the synchronous and asynchronous approaches are horizontally scalable. The scalability property of the solution is the result of several factors. First, because of the small-sized updates, all relevant change tables can readily fit into the memory of a single coordinator node. Second, the equi-join and update queries essential for the algorithm's implementation are horizontally scalable and can be readily integrated.The synchronous and asynchronous solutions fit several web application types. The former approach is suitable for web applications requiring consistent aggregate views while the latter algorithm can be applied for scenarios tolerating view staleness. However, the exact refresh frequency for the asynchronous algorithm is entirely application specific and needs to be further researched.

The consistency and fault tolerance of the aggregate view maintenance are guaranteed by the assigned global timestamps and two-phase commit protocol. In the worst case, when a node crashes the computation will be lost and simply restarted.

The proposed algorithm's efficiency can be increased by adopting base relation tagging for the incremental view maintenance as proposed in [5]. Thus, instead of performing join operations which may return zero rows, the coordinator will know beforehand which table rows participate in the views and decrease the number of join operations at the expense of additional storage. And finally, the non-distributive aggregate query support has been left for future research. In a naive implementation, the *MIN* and *MAX* functions need to inspect all table rows upon deletion. A possible solution would be to maintain distributed heap index structures for *top-k* queries and update them in the same transactional way as the view maintenance base tables.

Bibliography

[1] AGRAWAL, P., SILBERSTEIN, A., COOPER, B. F., SRIVASTAVA, U., AND RAMAKRISHNAN, R. Asynchronous view maintenance for vlsd databases. In *SIGMOD Conference* (2009), U. Çetintemel, S. B. Zdonik, D. Kossmann, and N. Tatbul, Eds., ACM, pp. 179–192.

[2] AGUILERA, M., DELPORTE-GALLET, C., FAUCONNIER, H., AND TOUEG, S. Stable leader election. *Distributed Computing* (2001), 108–122.

[3] AMAZON.COM. Amazon simpledb. Amazon.com, 2011.

[4] AMDAHL, G. M. Validity of the single processor approach to achieving large scale computing capabilities. In *Spring Joint Computer Conference* (1967), pp. 483–485.

[5] BAILEY, J., DONG, G., MOHANIA, M., AND WANG, X. S. Incremental view maintenance by base relation tagging in distributed databases. *Distrib. Parallel Databases 6*, 3 (July 1998), 287–309.

[6] BLAKELEY, J. A., AND MARTIN, N. L. Join index, materialized view, and hybrid-hash join: A performance analysis. In *ICDE* (1990), pp. 256–263.

[7] BORTHAKUR, D. *The Hadoop Distributed File System: Architecture and Design*. The Apache Software Foundation, 2007.

[8] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: a distributed storage system for structured data. In *7th USENIX Symposium on Operating Systems Design and Implementation* (2006), pp. 15–15.

[9] DEAN, J., AND GHEMAWAT, S. Mapreduce: simplified data processing on large clusters. In *6th Conference on Symposium on Opearting Systems Design & Implementation* (2004), pp. 10–10.

[10] GILBERT, S., AND LYNCH, N. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News 33*, 2 (June 2002), 51–59.

[11] GRAY, J., CHAUDHURI, S., BOSWORTH, A., LAYMAN, A., REICHART, D., VENKATRAO, M., PELLOW, F., AND PIRAHESH, H. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Min. Knowl. Discov. 1*, 1 (January 1997), 29–53.

[12] GRIFFIN, T., AND LIBKIN, L. Incremental maintenance of views with duplicates. In *SIGMOD Conference* (1995), pp. 328–339.

[13] GUPTA, A., HARINARAYAN, V., AND QUASS, D. Aggregate-query processing in data warehousing environments. In *VLDB* (1995), pp. 358–369.

[14] GUPTA, A., MUMICK, I. S., AND SUBRAHMANIAN, V. S. Maintaining views incrementally. In *SIGMOD Conference* (1993), pp. 157–166.

[15] GUPTA, H., AND MUMICK, I. S. Incremental maintenance of aggregate and outerjoin expressions. *Inf. Syst. 31*, 6 (2006), 435–464.

[16] LIBKIN, L., AND WONG, L. Query languages for bags and aggregate functions. *J. Comput. Syst. Sci. 55*, 2 (1997), 241–272.

[17] MEMBREY, P., THIELEN, W., PLUGGE, E., AND HAWKINS, T. *The Definitive Guide to Mongodb: The Nosql Database for Cloud and Desktop Computing*. Definitive Guide Series. Apress, 2010.

[18] MENASCÉ, D. A. Tpc-w: A benchmark for e-commerce. *IEEE Internet Computing 6*, 3 (2002), 83–87.

[19] MOORE, G. E. Cramming more components onto integrated circuits. *Electronics 38*, 8 (April 1965).

[20] QUASS, D. Maintenance expressions for views with aggregation. In *VIEWS* (1996), pp. 110–118.

[21] SRIVASTAVA, D., DAR, S., JAGADISH, H. V., AND LEVY, A. Y. Answering queries with aggregation using views. In *VLDB* (1996), pp. 318–329.

[22] STONEBRAKER, M., MADDEN, S., ABADI, D. J., HARIZOPOULOS, S., HACHEM, N., AND HELLAND, P. The end of an architectural era: (it's time for a complete rewrite). In *33rd International Conference on Very large data bases* (2007), pp. 1150–1160.

[23] WEI, Z., PIERRE, G., AND CHI, C.-H. Consistent join queries in cloud data stores. Tech. Rep. IR-CS-068, Vrije Universiteit, Amsterdam, The Netherlands, Jan. 2011. `http://www.globule.org/publi/CJQCDS_ircs68.html`.

[24] YU, Y., GUNDA, P. K., AND ISARD, M. Distributed aggregation for data-parallel computing: interfaces and implementations. In *ACM SIGOPS 22nd Symposium on Operating systems principles* (2009), pp. 247–260.