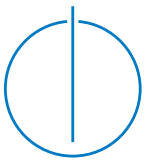
DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Benchmarking a View Maintenance System with help from TPC-H

Pranav Tomar



DEPARTMENT OF INFORMATICS

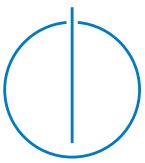
TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Benchmarking a View Maintenance System with help from TPC-H

Benchmarking eines View Maintenance Systems mit Hilfe von TPC-H

Author:	Pranav Tomar
Supervisor:	Prof. Dr. rer. pol. Hans-Arno Jacobsen
Advisor:	M.Sc. Jan Adler
Submission Date:	February 15, 2017



I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, February 15, 2017 Pranav Tomar

Acknowledgements

Abstract

Contents

1. Introduction

A Materialized view[1] is a set of query results that is pre-fetched and maintained over time so that the query can return immediately instead of waiting for results to be assembled at query time. In addition to allowing fast access, materialized views also abstract query complexity and allow clients to bridge the gap between storage schema and retrieval formats. Materialized views have been around for quite some time in SQL systems and most commercial relational databases provide built in support for them. As such the challenges associated with view maintenance are well understood in the relational context and traditional materialized views are considered a mature technology[2].

However, view management becomes an entirely new challenge at big data scale and in light of the NoSQL technologies that are used to cater to it. At its simplest, big data storage takes the form of key-value stores which typically store unstructured data in a distributed fashion. This allows for flexibility, speed, and scalability but puts the onus of constructing and maintaining consistent views on clients. This has opened a new field of research for view maintenance in large scale distributed databases and a number of projects like Apache Phoenix[3], Yahoo PNUTS[4], and Apache Casandra[5] now provide materialized views to some extent or the other. The View Maintenance System(VMS)[6] developed by Adler et al. is one such system which allows for maintenance of hundreds of views in parallel while also providing guarantees on consistency even under node crash scenarios. In this thesis we evaluate the performance of the VMS for different view types, under different workloads, and for varying number of clients. The basic performance metrics targeted are view update throughput and latency. We rely on the database benchmarking specification TPC-H[7] for generating the workload(basetable population) and a subset of TPC-H queries are used for configuring sufficiently complex views. This ensures that our VMS setup works with a data schema and using queries which are representative of a typical decision support system. Also given that the prototypical VMS instantiation works with HBase as its storage layer, we take advantage of the MapReduce framework in the underlying Hadoop system and use it as the workhorse for distributed workload generation.

The rest of this text is organized as follows. In chapter 2 we start off by trying to understand the VMS and its underlying technologies, and in addition examine relevant benchmarking and evaluation frameworks. In chapter 3 we analyze the original evaluation of the VMS by its creators and also try to draw inspiration from other efforts in the field of evaluation and benchmarking of NoSQL systems. In chapter 4 we design the evaluation driver. In chapter 5 we look at our implementation of the proposed evaluation driver. In chapter we use it to run experiments to evaluate the VMS. In

chapter 6 we conclude by retrospecting on our work and stating our achievements as well as shortcomings. Finally, in chapter 7 we propose how we can extend and improve the evaluation framework.

2. Background

In this chapter we begin by briefly discussing the factors that motivate a custom evaluation of the VMS. We then familiarize ourselves with the technologies underlying the VMS implementation, before looking at the VMS itself. We then discuss various evaluation and benchmarking strategies used for databases. Finally, we look at the Hadoop MapReduce framework since we intend to use it for distributed workload generation in our evaluation scheme.

2.1. Motivation

Benchmarks for evaluating and comparing traditional relational databases have been around for quite some time. A number of mature benchmark specifications exist which cater to different application domains like transaction processing or decision support. The Transaction Processing Council(TPC) leads the way with its popular benchmarks like TPC-C and TPC-H which have results published for most major vendors and database solutions available. Benchmarking and evaluation of big data systems on the other hand is a relatively newer challenge. Big Data benchmarking efforts need to not only cater to the much larger sizes(upto PB) of relatively unstructured data sets but also have to account for the distributed nature of the storage solutions. The absence of SQL interfaces is surely an advantage since the benchmarks don't have to worry about complex queries but instead there is a need to focus on scalability aspects like - elasticity and scale-out. TPC-DS is TPC's benchmark for SQL-based big data systems and acts as a sort of bridge between the worlds of traditional and NoSQL benchmarks. It addresses scale and variety, and relaxes ACID in favour of BASE(basically available, soft state, eventually consistent) constraints, but still expects an SQL interface. YCSB takes the first steps in the direction of NoSQL benchmarking by generalizing evaluation of cloud serving systems(specifically NoSQL key-value stores). It adds a scalability tier to the evaluation criteria. BigBench builds on top of TPC-DS and YCSB. It extends the TPC-DS data set to include unstructured data.

The View Maintenance System(VMS) that we have to evaluate is not exactly a data storage solution as will become clear in the next sections. But it is rather a distributed service that processes non-relational data distributed in HBase into relational views. Consequently its evaluation benefits from combining the scalability aspects of NoSQL benchmarks with the data model of TPC-H and motivates a customized approach which we discuss later in this text.

2.2. HDFS

Apache Hadoop is an open-source software framework for the distributed processing and storage of data at massive scales. It is made up of two components - the Hadoop Distributed File System(HDFS) and Hadoop MapReduce. HDFS is the distributed data storage service underlying Hadoop. It can be considered the open source sibling of Google's GFS. HDFS facilitates distributed processing(analysis or transformation) of large data sets using the MapReduce component and also forms the basis for distributed storage solutions like HBase. HDFS is not tightly knit into Hadoop which supports multiple other file system implementations like Amazon's S3 or MapR-DFS. However, as Hadoop's stock DFS it is the most popular choice.

HDFS[8], being a distributed file system, stores files by distributing(partitioning) them across many(thousands) of nodes. File metadata is stored on a dedicated server called the NameNode whereas file data itself is stored on other nodes called DataNodes. All servers are fully connected and communicate with each other using TPC-based protocols. Each file is partitioned into a number of related *blocks*, and blocks are replicated over multiple data nodes. A rack aware cluster ensures that blocks are replicated across racks so that data is not lost in case the entire rack fails. This ensures reliability and also has the added advantage of speeding up access times as well as allowing localization of computations to different nodes via MapReduce. The block size(typically 128MB) as well as the replication factor(3 by default) can be controlled on a per-file basis to make storage more efficient for different types of data. The metadata maintained at the NameNode represents the entire hierarchy of files and directores as inodes which record attributes like file permissions, access times, namespace and disk quotas. More importantly the metadata maps each block to its physical location on a DataNode. A read involves first asking the NameNode for the locations of the data blocks and then reading directly from the closest replicated blocks. Similarly, a write request also first contacts the NameNode to nominate a suite of DataNodes for replication and is followed by a pipelined write to these blocks. Periodic heartbeats are sent by the DataNodes to the NameNode to indicate successful availability. In addition these heartbeats also contain DataNode statistics like storage usage, storage capacity and the total number of transfers curretnly underway at the node. This allows the NameNode to make informed space allocation and load balancing decisions. The typical heartbeat interval is three seconds and DataNode timeout is ten minutes. If no heartbeat is received from a DataNode for over ten minutes then it is considered down and the NameNode makes it unavailable for reads or writes. At the same time it schedules the creation of new replicas for the blocks hosted by the faulty DataNode. As we can see the reads and writes are distributed across the DataNodes, but as the first point of contact the NameNode is a single point of failure. To prevent disruptions a secondary NameNode keeps a replica of the metadata so that it can be restored to the NameNode upon restart. User applications access the file system using the HDFS client which supports the usual operations to read, write and delete files and directories. The user references files using their filenames and directory paths and is generally oblivious to the location and replication of the underlying file blocks.

2.3. HBase

Traditional relational databases which have been around since the 1970s and are used by countless companies and organizations, are as relevant today as during the first years of their inception. Relational databases allow data to be stored in a structured way in tables with relationships between them which help model the real world significance of the data. In addition, constraints are used to help maintain the integrity of the data. Traditional databases also typically offer ACID properties and offer an SQL interface to the data. However, inspite of(and often because of) all these desirable characteristics, relational databases are not able to scale up well in face of massive data sets - the kinds generated by todays internet companies, IoT sensors, satellites, or particle accelerators. In the past companies could choose to keep the most recent or most relevant data and discard the rest thus keeping scales in check. But with the increasing use of machine learning and data mining techniques, data is now at the center of innovation and companies can no longer afford to loose it. Consequently all data being collected becomes important as a possible source of value generation through analysis and modelling. This means storage requirements have increased from GB to PB scales and data has become increasingly diverse and much more unstructured. To address storage and access of such large scales of semi-structured/unstructured data a lot of non-relational databases have come up. These are informally dubbed as NoSQL systems since most of them don't provide SQL but rather a simpler API-like interface to query data. A lot of these are simple "key-value" stores which store data in the form of a distributed, persistent, hashmap. HBase is one such key-value store. It can be considered an open source counterpart of Google's BigTable.

HBase[9] stores data in the form of non-relational tables. The tables are columnar which means that data is stored column-wise instead of row-wise. Each row has a key and cells can be accessed using this key and corresponding column name. Data is not typed and both keys and cells are treated as binary data. Table data is horizontally partitioned over the keyspace. Each partition is called a region and multiple *regions* are spread out across different servers called *Region Servers*. The system can be scaled up by adding Region Servers. A central component called a *Master* controls the mapping between the keyspace and the Region Servers and can help balance system load and distribute data uniformly. Clients first contact the Master to copy the mapping of the keyspace to Region Severs and then all subsequent reading/writing takes place directly with the Region Servers. A backup Master(on a different node) acts as an active standby to take over in case the primary goes down. Columns can be grouped into *column families* and all columns in a family are stored together in the same low-level storage files, called *HFile*. The HFiles are stored in the HDFS data node collocated with each Region Server. At each Region Server data is first written to a commit-log called *Write Ahead Log(WAL)* and then added to an in-memory write-buffer called *Memstore* before being flushed to HFiles. In addition HBase maintains an in-memory read-cache called the *Block Cache*. Horizontal partitioning allows fast access to billions of rows and the underlying HDFS replication affords resilience. The simple storage model of HBase is extended by the *View Maintenance System(VMS)* which we look at next.

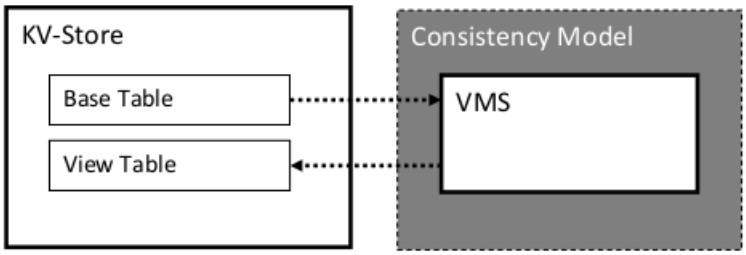


Figure 2.1.: VMS System Overview[6]

2.4. View Maintenance System

2.4.1. Introduction

Distributed key-values stores like HBase are currently the solution of choice for many data-intensive applications. They are deployed as large-scale highly-distributed systems which allow horizontal scaling by partitioning data and request load across a configurable number of nodes. Moreover, these systems have high availability, fault tolerance, and load balancing built in by design. However, to achieve these properties at scale KV-stores sacrifice an expressive query language and data model, and only offer a simple API comprising of *get*, *put*, and *delete* operations. This design provides efficient simple single row access but imposes a challenge for applications that require even slightly more complex query capabilities.

The *View Maintenance System(VMS)*[6] developed by Adler et al. is a distributed system that addresses this by providing incrementally maintained views on top of a typical KV-store. It gives sophisticated query capabilities to applications which can now define views corresponding to complex queries which are otherwise either impossible in the KV-store or impractical for the application to construct on its own. The VMS materializes these views as tables in the KV-store and so all table characteristics like concurrent access, availability, and fault-tolerance apply to views as well. To understand its operation(see figure 2.1) the VMS can be described as a system which efficiently propagates base table updates to corresponding view tables. Moreover, its distributed design(which we look at next), allows it to do this in a scalable as well as parallelized manner. The prototypical instantiation of the VMS which uses HBase as its underlying KV-store is the system under evaluation(SUT) in this thesis.

2.4.2. Design Overview

We first look at the generalized design of the underlying KV-store before analyzing the VMS itself. The upper half of figure 2.2 shows the the system *nodes* of the KV-store. Each of these is responsible for storing a part of the data and as such forms the unit of scalability. A *file system* builds the persistence layer of a node. KV-stores organize data into *tables* which need not follow a fixed schema. Each table consists of *rows* of records. Each row can hold a variable number of *columns*, and columns

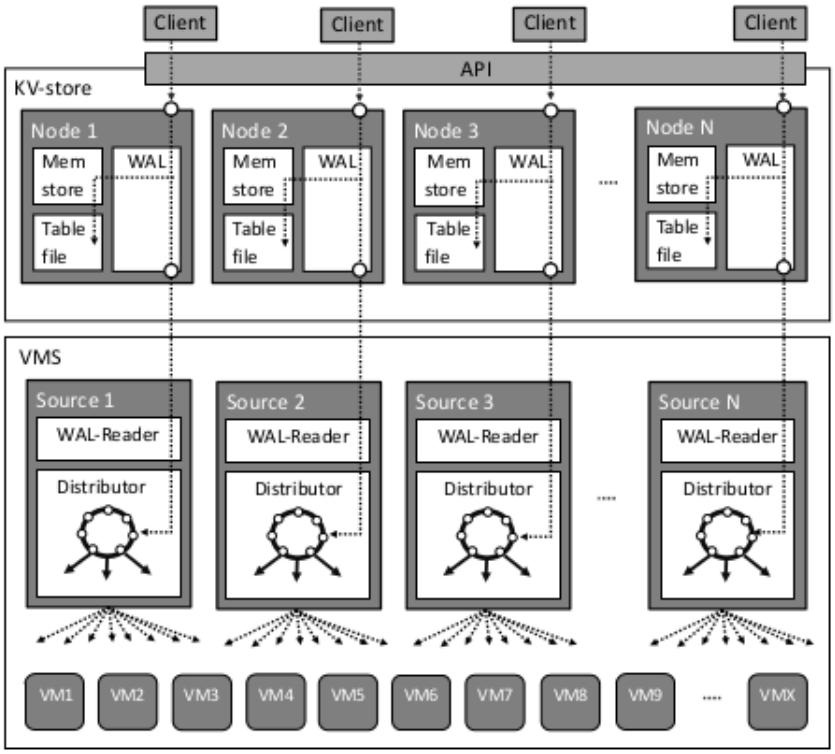


Figure 2.2.: KV-Store and VMS[6]

can be grouped into *column families*. Table data is partitioned across multiple nodes using *key ranges*. In the case of HBase each table partition is called a *region* and so the nodes hosting the table data are called *region servers*. Multiple regions can be assigned to a region server. Also, regions can be split or moved between region servers to balance load or to achieve uniform distribution of data. HBase uses HDFS as its persistence layer and so each region server is collocated with a HDFS DataNode. System management tasks are performed centrally by a special node which could be either designated explicitly(HBase, BigTable), or elected(Cassandra), or vary on a per record basis(PNUTS). In case of HBase this node is called the *master* node. KV-store API supports only three operations: *get*, *put*, and *delete*. Each of these is routed to the node responsible for the related key range. Every node maintains a *transaction log*(TL) file and an in-memory write-buffer called *memstore*. Client operations return successfully after writing to the TL. The operation is then inserted into the memstore which is flushed to disk after growing to a certain size. Each flush to disk generates a set of *table files* which is how the table data is actually stored on the underlying file system. Reads go through a read-cache called *block cache*.

The VMS is loosely coupled with the KV-store. It only interacts with the KV-store by reacting to two different types of events: *administrative events*(like node addition/deletion) and *data events*(like base table update). KV-stores different ways to react to administrative events. HBase allows developers to define *coprocessors* as admin event

callbacks which are used to notify the VMS to take appropriate actions. The VMS processes data events in a simple and decoupled manner by asynchronously monitoring the TL file for updates through its WAL-Reader component.

We finally look at the VMS design. It comprises of n VMS *sources*(figure 2.2), each connected to a KV-store node on one side and multiple *view manager*(VM) nodes on the other. Each source is collocated with a KV-store node and is responsible for propagating base table updates to *view managers*(VM). The WAL-reader reads the WAL for new base table operations and passes them to the *distributor* component which in turn distributes them to the VMs. A number of VMs can be registered with a distributor and since they are lightweight and stateless they can be easily added, removed, or moved around across the sources to accommodate changing loads and VM failures. Each VM caters to a subset of the keyspace of each view which is decided by the distributor using consistent hashing. We next look at how data flows in the VMS.

2.4.3. Update Propagation

The primary goal of the VMS is to propagate base table updates to view tables in an efficient manner. This means that negligible overhead should be added to the functioning of the KV-store and reads and writes to KV-store tables should have latencies comparable to a system which doesn't have a VMS running on top of it. To achieve this a source distributes the arriving base table updates to the VMs via consistent hashing by maintaining a hash-ring over the keyspace of each view, where active active VMs are registered. An incoming base table update is then hashed into the ring using their corresponding view table row-key and associated in clock-wise direction with active VMs. This distributes operations across the available VMs. Hashing the updates using view table row key instead of base table row key is an essential part of ensuring consistency in this highly parallelized environment as we see in the next section. To access and update the views the VM acts as a client to the KV-store using its standard client API.

2.4.4. Consistency

In context of the VMS, consistency implies correctness of the view tables in face of the incremental updates that are constantly applied to them. Since these updates are applied in a distributed and parallelized manner there is possible room for divergence between the base tables and corresponding view tables. The *strong consistency* model enforced by the VMS ensures this doesn't happen. Updates applied to the base table take the table from an initial state B_0 to a set of intermediate states B_i and finally an end state B_f. Similarly, the view table is updated from V_0 to V_f via a set of intermediate states V_i. Consistency can be defined by the following levels:

- *Convergence:* After all the updates have been applied B_f and V_f are in agreement.
- *Weak consistency:* The view converges and each intermediate view table state V_i is in agreement with some intermediate base table state B_i.

- *Strong consistency:* Weak consistency is achieved and in addition pairs of agreeing intermediate view and base table states are in the same sequence i.e. $V_i \leq V_j$ implies $B_i \leq B_j$ and vice verca.

The following three requirements are sufficient to guarantee that views are strongly consistent:

1. *View updates are applied exactly once*
2. *View updates are atomic*
3. *(Base)record timeline is always preserved*

Updates are assigned a unique globalId. VMs track the highest globalId that they have seen and ignore any update with a lower ID as a duplicate. This combined with the fact that updates can not be lost since WAL is replicated in the file system, ensures exactly once semantics of requirement 1. Row operations in the KV-stores are atomic and since only one VM is responsible for a view table row key any combination of different row operations will also be atomic as required for the second condition. Finally, distribution of updates according to hash of the view table row-key creates a record timeline which represents the base table timeline as long as updates don't modify the view table row-key. In case they do then the effect is mitigated by using a special update buffer. Thus requirement 3 is also satisfied, and the VMS manages to provide strong consistency.

2.4.5. View Maintenance

Auxiliary Views

Auxiliary views are hidden views in the VMS which act as the building blocks for other views. They are required to both facilitate and speed up view maintenance. . *Delta*, *Pre-Aggregation*, and *Reverse Join* are the three kinds of auxiliary views used. A delta view keeps tracks of base table changes between successive update operations. Pre-Aggregation views prepares for aggregation by sorting and grouping base table rows. Subsequently, aggregation views only need to apply their aggregation function to the pre-computed state. A reverse-join view supports the efficient and correct materialization of joins in the VMS.

Standard Views

The VMS provides *selection, projection, aggregation, index, and join* as standard views. A selection view selects a set of the base table rows based on a selection condition. A projection view selects a set of columns from the base table. In an aggregation view records identified by an *aggregation key* combine into a single view table record with the aggregation key as its row key. Index views provide fast access to arbitrary columns by using those columns as the row key. Finally joins combine multiple tables using a common column(join key), with the help of reverse-join auxiliary views.

2.5. Evaluation Frameworks

2.5.1. TPC-H

The TPC-H specification document comprehensively covers all that is required for implementing the benchmark, however, from the point of view of general understanding it is a bit disorienting in its details. The excellent paper by Thanopoulou et al.[10] gives a much better general description of all the steps required to implement the benchmark and so this section draws almost exclusively from it instead.

A typical business enterprise makes use of well-structured databases to facilitate and track operations. These databases record business actions(like sales transactions, procurement etc.), participating actors(like clients, partners, suppliers etc.) and items(like inventory, equipment etc.). The queries that are executed can be classified into Online Transaction Processing(OLTP) and Decision Support(DSS) ones. The former are comprised of basic information retrieval and update operations like customer look-up, registration, or update. The latter are meant to support business decisions by analyzing historical data and are consequently more complex and long-running compared to OLTP queries. DSS queries can be further classified as ad-hoc and reporting queries. Ad-hoc queries are executed ad-hoc - i.e. whenever a business analyst/executive needs to find an answer. Reporting queries are less frequent and typically scheduled to help businesses monitor and control operations on a periodic(monthly, weekly etc.) basis.

The Transaction Processing Council Benchmark^{TM}H (TPC-H)[7] is a benchmark for decision support systems - database systems running ad-hoc DSS queries. These systems run complex queries on large volumes of historical data to give answers to critical business questions. TPC-H models a business database and provides a suite of business oriented ad-hoc queries and concurrent data modifications to run on it. Both the data model and the queries have been chosen such that they are broadly representative of real decision support systems but are still sufficiently easy to implement. The benchmark focuses on the system utilization and complexity of operations in a decision support system while at the same time reducing the diversity of operations. This allows for a common ground for benchmarking and comparison of systems operating at different scales. As such the TPC-H benchmark has been used extensively by database vendors to show their product's performance as well as by database researchers for validating their approaches.

The specification can be broken down into two parts - (i) the TPC-H Database Schema and (ii) the TPC-H Workload. The TPC-H data schema[7] consists of eight base tables(see appendix A.1) connected to each other by primary key-foreign key constraints. It is meant to mimic the activity of a typical product retailer that manages, sells, or distributes products world wide. The tables nation and region have a fixed size, but all others scale in size proportional to a constant scale factor(SF), as seen in table 2.1. The scale factor determines the size of the database in GB and valid scale factors are: 1, 10, 30, 300, 1000, ..., 100000. So at a scale factor 10 the test database has a total size of 10 GB. TPC-H also provides a tool for synthetic data generation called DBGEN which can be used to populate the test database at different scales. In addition

Table Name	Number of Rows
Region	5
Nation	25
Supplier	10,000 x SF
Customer	150,000 x SF
Part	200,000 x SF
Part_Supp	300,000 x SF
Orders	1,500,000 x SF
Lineitem	6,000,000 x SF

Table 2.1.: Number of rows per table in the TPC-H database relative to the scale factor. The two largest tables are Lineitem and Orders and hold about 83% of the total data.[10]

to generating data for load, the tool also generates data for updates - keys for deletion and new rows for insertion. It is also important to note that from the point of view of TPC-H the DSS database is different from the transactional OLTP database. It is not directly updated by real time business activity but instead is updated through a set of refresh functions which push data from the live-OLTP database to the analytical DSS one (see figure 2.3).

The TPC-H workload[7] consists of 22 queries(see appendix A.2) and 2 update procedures(see appendix A.3). The queries model frequently asked decision making questions in areas such as pricing and promotion, supply and demand management, profit and revenue management, customer satisfaction survey, market share survey and shipping management. From a technical standpoint the queries encompass a wide diversity of operations, work on large volumes of data, and generate high disk and CPU activity. The two update procedures are called refresh functions(abbreviated as RF) and represent periodic data refreshes - one being responsible for inserting new orders(RF1) and the other for deleting old orders(RF1). The workload is applied in two different ways- sequentially and concurrently, as part of two different tests, to simulate both single-user and multi-user scenarios. TPC-H also provides a tool for generating executable queries called QGEN. The tool converts the query templates provided in the specification to executable queries by plugging in appropriate values for certain substitution parameters. At the same time it also ensures that the parameters are generated in such a way that the performance of a query with different parameters is comparable. The refresh functions are provided only in pseudo-code and we are responsible for generating the corresponding database operations(inserts/deletes) ourselves by using update parameters generated by DBGEN. It is important to note that the original dataset generated is 75% sparse for the lineitems and orders tables. This is done by populating only the first 8 values in each group of 32 keys thus leaving 3/4ths of the keys empty. For the first 1000 iterations RF1 is responsible for inserting 0.1% new rows with primary keys in the second 8 key values and RF2 is responsible for deleting 0.1% of existing rows with primary keys in the original first 8 key values of

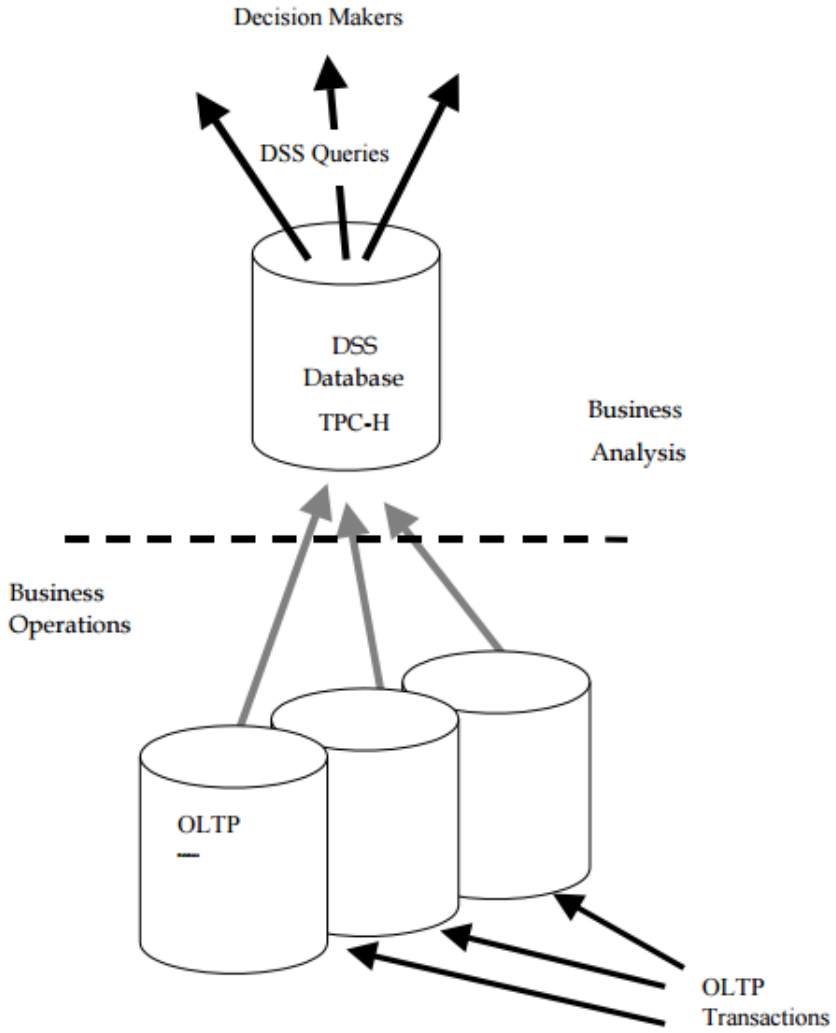


Figure 2.3.: TPC-H Business Environment[7]

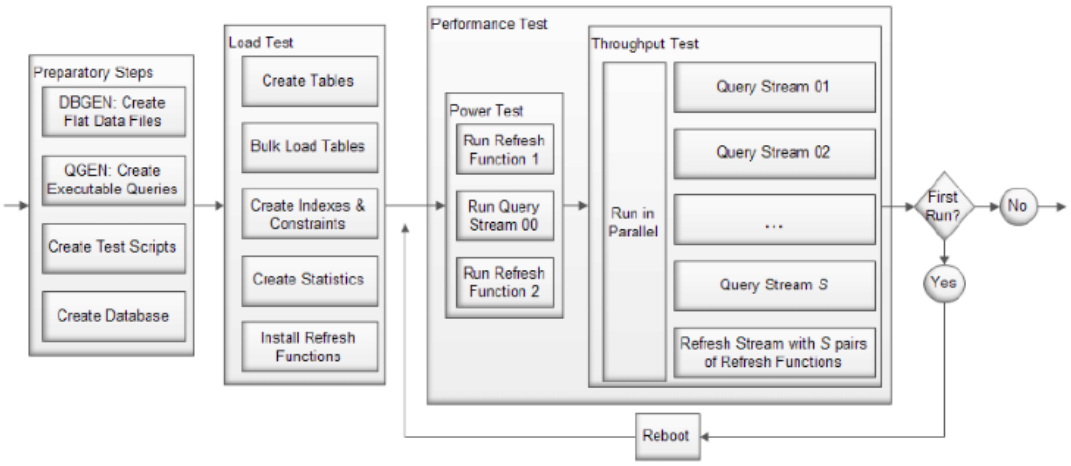


Figure 2.4.: TPC-H Execution[10]

each group of 32 keys.

Each benchmark execution is divided into a Load Test followed by two executions of the Performance Test(which are averaged)(see figure 2.4). The performance test is further subdivided into a Power Test and a subsequent Throughput Test. A query stream is a sequential execution of each of the 22 TPC-H queries in a random order, while a refresh stream is a sequential execution of a number of pairs of refresh functions. The Load Test is the phase of the execution where the database is originally loaded. It excludes preparatory steps like initial table creation and generation of flat data files using DBGEN. The Power Test comprises of a sequential execution of RF1(inserts) followed by a query stream and finally execution of RF2(deletes). It aims at measuring the raw query execution power of the system with a single user. The subsequent Throughput Test consists of a parallel execution of a minimum number of query streams and one refresh stream. The minimum number of query streams(which is supposed to be equal to the number of pairs of refresh functions in the refresh stream) is decided by the scale factor(See Figure 3). The throughput test is meant to measure the ability of the system to process the maximum number of queries in the least possible time, possibly taking advantage of any parallelism in the system or the underlying computer architecture. In other words it measures system performance in the face of a multi-user workload.

The benchmark reports two primary metrics - the composite query-per-hour performance metric(QphH@size) and the price/performance metric(Price-per-QphH@Size). The first is simply the geometric mean of the Power@Size and the Througput@Size metrics which are obtained from the Power Test and Throughput Test respectively. The second measures price per unit performance by simply dividing the total dollar cost of the system by the first metric. The throughput test metric Throughput@Size measures how many total queries were executed in an hour and is obtained by dividing the dividing the total number of queries by time in hours and finally multiplying with

scale factor to account for system scale. The power test metric Power@Size measures the number of queries executed per hour at a given scale and is obtained by dividing the geometric mean of all query and refresh function execution times divided by total time and then multiplied by the scale factor. The formulae for the different TPC-H performance metrics are listed in appendix A.4.

2.5.2. TPC-DS 2.0

TPC-DS is the first industry standard benchmark for measuring the end-to-end performance of SQL-based big data systems like Spark and Hive[11]. It specifies a rich database schema which supports realistic scaling upto 100TB on clustered systems. In addition it specifies a large number of very complex queries that cover the entire width and breadth of a decision support system. The minimum raw database size is increased from 1GB in older benchmarks like TPC-H to 1TB. However big data systems operate at even bigger scales and TPC is currently evaluating PetaByte scaled data sets. Big data systems are essentially BASE(Basically available, soft state, eventually consistent) systems[12] and so unlike TPC-H, TPC-DS doesn't enforce ACID(atomicity, consistency, integrity, durability) compliance on them. Instead it requires that the system under test continue executing queries and data maintenance functions with full data access even during irrecoverable failure of a single disk(or node). This makes durability not only a functional requirement but also part of the performance test. TPC-DS also separates the querying of data from data maintenance because their overlap requires ACID support. This means that the refresh functions are moved out of the throughput test and into a separate maintenance test of their own. This simpler approach fits many big data use cases which focus on data analysis while providing separate data refresh windows. The benchmark evaluation thus consists of the usual load test followed by a power test which is then followed by two pairs of throughput test-maintenance test sequences.

An important aspect of TPC-DS is that it combines both ad-hoc and reporting queries in the same benchmark[11]. An ad-hoc workload models users sending impromptu queries to the system and consequently can't be optimized for. On the other hand queries in the reporting workload are well known in advance and so the DBA can optimize the system specifically for these queries and achieve fast execution by using clever data placement methods(e.g. partitioning and clustering) and auxiliary data structures(like indexes and materialized views). TPC-DS achieves to combine these two query types by dividing the data schema into two distinct reporting and ad-hoc parts. Iterative OLAP queries(queries grouped together) and data mining queries are two other types of queries in addition to the ad-hoc and reporting queries which are comprehensively covered by the 99 queries in the TPC-DS specification. Another point of difference from earlier benchmarks is its snowflake schema which allows data to be scaled up realistically. Earlier benchmarks like TPC-H scale up to 100 TB as well but the data is not realistic. The snowflake schema divides the data in to 7 fact tables and 17 dimension tables which cover three different sales channels between them. TPC also provides tools(DSDGEN and DSQGEN) for generating synthetic data and executable queries.

2.5.3. YCSB

Moving away from the domain of traditional relational databases, we now look at cloud serving systems - systems that offer online read/write access to data. Many of these are also referred to as "key-value stores" due to their simple storage and access mechanisms or "NoSQL systems" due to the absence of any SQL interface. These systems favor low latencies and elasticity over complexity. Many of these systems like Yahoo!s PNUTS, Google's BigTable, and Amazon's Dynamo drive the big Internet companies of today. However, a large variety makes it difficult for devleopers to choose the appropriate system. The obvious differences in data models like column-oriented BigTable model of Cassandra or HBase vs the simple hashtable model of Voldemort can be be documented and compared qualitatively. Performance comparison is a much harder problem. Different systems make different kinds of tradeoffs in order to optimize for different applications. The main tradeoffs are[13]: read vs write performance, latency vs durability, synchronous vs asynchronous replication, and data partitioning. Moreover there are a bunch of system parameters like partitioning, replication factors, cache sizes, and consistency parameters that can all be tuned and impact performance in different ways for different systems. In light of these problems it becomes evident that benchmarking these much simpler systems is more difficult compared to the traditional relational systems that we have inspected before.

Yahoo!'s Cloud Serving Benchmark (YCSB)[13] is an open source framework that addresses this problem. It allows for an apples-to-apples performance comparison between various types of data-serving systems including but not limited to NoSQL data stores like HBase, Cassandra, Redis etc. The framework consists of a workload generating client and a package of standard workloads that cover interesting parts of the performance space (read-heavy work-loads, write-heavy workloads, scan workloads, etc.). An important aspect of the YCSB framework is its extensibility: the workload generator makes it easy to define new workload types, and it is also straightforward to adapt the client to benchmark new data serving systems.

YCSB is divided into two tiers[14]- the first focuses on performance and the second addresses scalability. The *performance tier* measures the latency of requests when the system is under load. It tries to reveal the tradeoff between latency and throughput for the given system setup. As throughput increases(increased load) the system resources are more burdened and consequently latencies increase. The aim is then to find out what is the maximum throughput that the system can support for any acceptable latency. The *scaling tier* examines the system behaviour as more machines are added during operation. *Scalup* and *Elastic-Speedup* are the two metrics measured in this tier. *Scaleup* judges how the database performs as the number of machines are increased whereas *Elastic-Speedup* measures system performance as the number of machines are added during operation. If the database system has good scaleup properties, the performance (e.g., latency) should remain constant, as the number of servers, amount of data, and offered throughput scale proportionally. A system that offers good elasticity should show a performance improvement when new servers are added, within a short period of re-configuration.

2.5.4. BigBench

The emergence of the first generation of commercial database systems in the 1980s gave rise to the need for well defined benchmarks that allow measurement and comparison among different vendors. The Transaction Processing Performance Council(a consortium of database vendors and academic organizations) stepped in to fill this gap with a series of data warehouse end-to-end benchmarks like TPC-D in the 90's followed by TPC-H and TPC-R in the early 2000's. These benchmarks evaluated single as well as multi-user performance of complex queries on a data warehouse along with concurrent updates but were restricted up to the terabyte scale. However, with the advent of big data scale applications driven by the internet, data volumes soon moved towards the petabyte range. There was now a need for newer benchmarks to support the large volumes of fast moving and diverse data sets. To address this TPC created the TPC-DS benchmark. There were also other efforts from academia as well as industry like Yahoo's cloud serving benchmark - YCBS, which aimed at evaluating NoSQL data stores. However, all these benchmarks have been either lacking big data characteristics or are micro or component benchmarks which do not evaluate the system as a whole.

BigBench[15] is a big data benchmarking specification developed in collaboration between academia and multiple hardware and software vendors. It builds upon and borrows elements from previous big data benchmarking efforts like YCSB and TPC-DS. The specification is also available as an open source implementation kit developed together by Cloudera and Intel. As pointed out in [16] it has a number of advantages because of being specification-based and open source. The formalism and flexibility provided by the specification allows for multiple implementations to co-exist and reused, and the open source implementation *kit*[17] lowers the entry-barrier for benchmarking big data systems.

The BigBench specification consists of two parts - the data model and the workload specification. The data model specification[18] is a mix of structured, semi-structured, and unstructured data. The structured part of the data is adopted from TPC-DS and models an online product retailer, the unstructured part consists of user-reviews, and the semi-structured data is in the form of web logs of user-clicks. Consequently the workload specification too a combination of a separate set of queries, one for each part of the data model. The queries for the structured part are borrowed from TPC-DS[11]. The queries for the semi and un-structured parts of the model are borrowed from a McKinsey report on Big Data use cases and opportunities. In addition to the queries, the initial load is also considered as part of the workload and is called Transformation Ingest(TI)[15]. The BigBench specification takes care of all the three aspects of big data systems - the mixed nature of the data model ensures *variety*, *velocity* is ensured by periodic refreshes interspersed within possibly concurrent parallel query streams, and *volume* is addressed by data generation at different scales using an extended PDGF parallel data generator[15][17]. The reported metrics focus around query execution times and can be reported either end to end for the execution pipeline or for each individual query/query stream. The benchmark can also simulate multiple users by allowing for parallel workload streams.

2.6. Hadoop MapReduce

Hadoop MapReduce[19] lies at the heart of the Hadoop system. Building on top of the distributed storage infrastructure provided by HDFS it provides an environment for parallel processing of large scale data in a fault tolerant, scalable, and distributed manner. An important feature of MapReduce is the ability to move computation to data so as to minimize network bandwidth utilized for remote computations. This is what makes the parallel processing truly parallel. Hadoop MapReduce is an open source implementation of the general MapReduce algorithm proposed by Google.

A MapReduce execution consists of a set of parallel map tasks followed by a set of parallel reduce tasks. Each map task runs on a chunk of the input data and produces some output, the reducers then collate the outputs from different map tasks to generate a final result which is written back to HDFS. The MapReduce framework consists of a single master JobTracker and one slave TaskTracker per cluster-node. The master is responsible for scheduling the jobs' component tasks on the slaves, monitoring them and re-executing the failed tasks. The slaves execute the tasks as directed by the master. Typically the compute nodes and storage nodes are collocated. This means that the map tasks being run on a compute node under the supervision of a TaskTracker all run on the data chunks residing on the node's local HDFS DataNode. This gives a very high aggregate throughput across the cluster. Scalability is achieved by simply adding new nodes. Applications specify the input/output locations and supply map and reduce functions via implementations of appropriate interfaces and/or abstract-classes. These, and other job parameters, comprise the job configuration. The Hadoop job client then submits the job (jar/executable etc.) and configuration to the JobTracker which then assumes the responsibility of distributing the software/configuration to the slaves, scheduling tasks and monitoring them, providing status and diagnostic information to the job-client.

2.7. Understanding MapReduce Parallelization

MapReduce programs are designed to compute large volumes of data in a parallelized fashion. This is achieved by breaking the data into small parts and processing many of these parts concurrently on different nodes. The processing is done in two steps(or phases) called Map and Reduce. The input data is treated as a list of entries, with each entry itself being treated as a key-value pair. The job of the map phase is to transform(map) each of these key-value pairs into a new set of key-value pairs. This means that for each key-value entry in the input, zero, one, or more key-value pairs would be generated as output. In the Word Count example in figure 2.5, this is a one-on-one mapping from <sequenceNumber, word> to <word, 1>. This intermediate output is then shuffeled and sorted by key - i.e. the entries with the same key(word) are sent to the same node where they are sorted(or grouped) by key. This ensures that all entries with the same key are reduced at the same node. In case of the Word Count example the reduction step involves aggregating the count of entries with same key(word) to yield the word count.

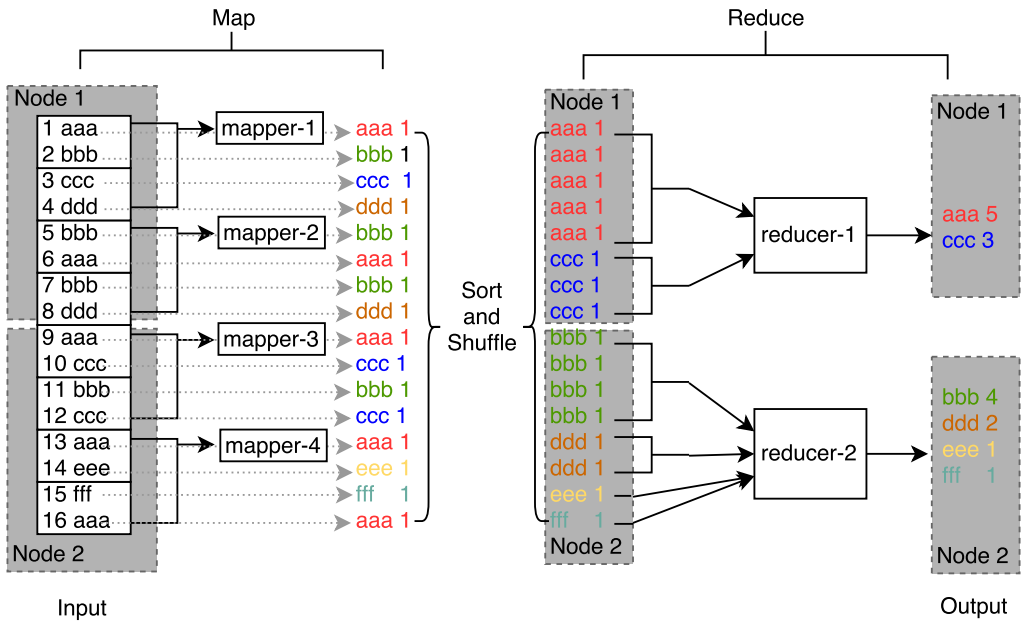


Figure 2.5.: Word Count in MapReduce

A Hadoop invocation of a MapReduce program is called a Job. The map phase of a job consists of multiple map tasks(or mappers) which work on one or more input file parts which are assigned to them. For example in the Word Count example mapper 1 works on the first two parts which cover keys 1 to 4, mapper two works on the next two parts(keys 5 to 8), and so on. Each node has multiple mappers but the number of file parts is usually much more than the number of mappers. Therefore each mapper works on a number of parts one after the other. The processing of items within a part is also sequential. The map phase is considered complete when all parts have finished processing.The reduce phase consists of a number of reducer tasks. Each node can host a number of reduce tasks. The output from the map phase is sorted by key and then shuffled across the network so that all the items with the same key land at the same reduce task. This is important because the reduce tasks generate output on a per key basis.

By default each hadoop node runs 2 mappers and two reducers, but this can be configured using the properties:

```
mapred.tasktracker.map.tasks.maximum
mapred.tasktracker.reduce.tasks.maximum

or programmatically in the Job driver:

jobConf.setNumMapTasks(int num)
jobConf.setNumReduceTasks(int num)
```

Picking the right number of tasks is critical to achieve optimal concurrency in the

system. Increasing the number of tasks increases the framework overhead, but also increases load balancing and lowers the cost of failures. At one extreme is the 1 map/1 reduce case where nothing is distributed. The other extreme is the 1 million maps and reduces where the framework runs out of resources for the overhead. A good starting point is to have the number of mappers/reducers equal to the number of cores per node. For less CPU-intensive tasks it can be increased to 10-100 mappers per node. It is also important to note that task setup takes a while so the tasks should take at least a minute to execute. Also, at most a task must take no more than ten minutes for fear of timing out. The length of the task in general depends on the size of the input part it operates. By default a part is 64 MB but can also be configured using the following property:

`mapred.max.split.size`

We have now seen how MapReduce splits work across both space and time and we conclude with the observation that the degree of parallelization is cofigurable depending on the task at hand and the available system resources.

3. Related Work

The most important related work is the original evaluation of the View Maintenance System by its creator. For this we look at the evaluation section of Jan Adler's paper on Dynamic Scalable View Maintenance[6]. We also look at the paper on Yahoo Cloud Serving Benchmark by Cooper et al. to get more insight into evaluation of distributed serving systems.

3.1. VMS - Original Evaluation

The System Under Evaluation (SUE) in this thesis is essentially the same as the original VMS implementation, except for its ability to handle more complex view types. It is an implementation of VMS in Java running on top of Apache HBase. We now look at the experimental setup used in its original evaluation. All experiments were performed on a cluster of 40 nodes, each running Ubuntu 14.04. Out of these approximately a fourth are used for HDFS (v1.2.1) and HBase(v0.98.6.1). One of them hosts the HDFS NameNode and HBase Master whereas the remaining host an HDFS data node along with an HBase RegionServer. VMS View Managers are deployed on half of the total nodes in the cluster. The remaining one-fourth are used to host HBase clients to generate the workload.

Prior to each experiment, an empty base table is created along with a set of view tables. The base tables are pre-split into 50 regions. This allows HBase to control the regions with high granularity and ensures a balanced distribution of keys in among the regions servers. For aggregation views the base table consists of two columns - one column with the aggregation key, and the second with the aggregation value. For selection views again two columns are used, with the selection condition applied to the second column. For the join views two base tables are required with different row keys. The row key of one of the tables is stored in a column for the other one and is called its foreign key. The workload consists of insert, update, and delete operations issued to HBase using its client API. The operations are generated according to two different key distributions - uniform and zipfian. The former spreads out the distribution uniformly across the key space, whereas the latter concentrates it on a few "hot-spots". The primary evaluation criteria are throughput and latency of view maintenance operations. The following different types of experiments are conducted.

Impact of view type on throughput

In this experiment, view maintenance throughput (updates per second) is measured for each different view type separately. Selection is tested with three different selectivity

levels - selection of 10%, 50%, and 90% of base table records. Compared to other views selection has the best performance, and throughput is actually limited by how fast clients can push updates to the system. Furthermore, increasing the number of VMs only slows the system down as more concurrency is introduced. Aggregation is tested for count, sum, min, max, and index view types. Since aggregation views require auxiliary view maintenance their throughput is lower. However, unlike selection, aggregation benefits from increasing the number of VMs. Finally, join views show the least throughput compared to the other two types. This is because joins require the maintenance of more complex auxiliary tables. Also, similar to aggregation, joins also benefit from increasing the number of VMs.

Cost and benefit of view maintenance

In this experiment, the benefit of view maintenance is brought forward by comparing it to two alternate approaches - client table-scan and server table-scan. The count aggregation view is used for this comparison. In client table-scan, a client scans the base table and aggregates all the values on its own. In server table-scan, a client sends a request to HBase which internally computes the view and returns it as an output. Now since client table scans are sequential in nature and require a large number of client calls to HBase, this approach is impractical, especially with increasing table sizes. In other words, it is not feasible at scale for a client to maintain views on its own. Server table-scans are faster since they are parallelized by HBase, making use of MapReduce architecture under the hood. However, it still takes a few seconds for aggregating count for a table with 1 million rows. Compared to both these approaches, lookup from a VMS maintained view is instantaneous, since it is just a table lookup. This fast access comes at a cost of more resource consumption due to view maintenance activities, which in turn increases update latencies. So a base table load takes about 40% more time when view maintenance is in effect. However, this increase in latency is acceptable in light of massive gain in view lookup speeds.

System scalability

In this experiment, the number of RegionServers is increased and system performance is observed. A near linear increase in throughput is observed with increase in number of region servers. Moreover, this increase is independent of view type. This is because adding a new RegionServer involves adding a new node and consequently leads to more I/O bandwidth (a new disk). This improves overall HBase performance which means both client requests and view table updates can be made faster to HBase, thus improving overall throughput.

Impact of data distribution

In this experiment, the impact of different workload distributions on performance is assessed. The number of VMs are varied and latencies are recorded for each distribution. For uniform distribution, latencies are independent of the number of VMs assigned

as even a small number of VMs are sufficient to handle a spread out load. For a Zipf distribution, however, latencies are much higher. This is because the workload is concentrated("hot-spotted") in a few region servers. In this case there is a benefit from addition of more VMs to the affected region servers. This highlights the importance of dynamically assigning VMs based on RS load to optimize performance. However, another point that comes forth is that in case of a "hot-spotted" workload the bottleneck might actually be HBase since only a few region servers are loaded and this can not be offset even with addition of new VMs.

Impact of VM Crash

In this experiment, view maintenance latencies are examined in the face of VM crashes. 20 VMs are distributed evenly across 10 region servers. One VM is aborted for each region server and latencies go up due to the additional time required for recovery. Overall, it is observed that the impact of VM crash lessens with increasing number of VMs, as should be expected.

3.2. YCSB Insights

We overviewed the Yahoo! Cloud Serving Benchmark (YCSB) in the background material in section 2.5.3. We now look more closely at the two aspects which make it a significant benchmark for cloud serving systems. We benefit from this exploration because the SUE in this thesis exhibits characteristics of cloud serving systems like K-V stores, not in the least because it is a similar distributed system, but also because it is build on top of one such K-V store(HBase), and consequently needs a similar treatment for workloads.

Benchmark Tiers

YCSB breaks down the evaluation of the targeted K-V stores into two tiers - viz. the *performance tier* and the *scalability tier*. The *performance tier* looks at the latency of requests when the database is under load. As the load (and consequently throughput) increases the latency of individual requests also increases since there is more contention for system resources. The aim of the performance tier, then, is to find the best tradeoff between latency and throughtput - i.e. finding the maximum load the system can handle at a reasonable latency.

The *scalability tier* acknowledges the distributed nature of the system being evaluated, and examines the impact on performance as more machines are added to the system. There are two metrics which are measured in this tier, viz. *scaleup* and *elastic speedup*. Scaleup measures how the system performs as the number of machines increases. Performance is measured with increasing loads supported by an increasing number of machines. For a system with good scaleup properties the performance is expected to remain constant as the increasing loads(and throughputs) are offset by the increased number of machines.

Benchmark Workloads

The YCSB framework includes a *core package*. This contains a collection of related workloads. Each workload targets a different section of the database performance space by varying mix of read/write operations, data sizes, and request distributions. This allows a direct examination of different tradeoffs made in different systems - like optimization for reads vs writes, or inserts vs updates.

Four operations make up the workload - viz. insert, update, read, and scan. Insert inserts a new record, update updates a record by replacing the value of one field. Read involves reading either one or all fields of a record. Scan involves scanning a fixed number of records in order starting at a given record key. Furthermore, these operations are distributed in the keyspace of the corresponding base table using four different types of distributions - viz. uniform, zipfian, latest, and multinomial. Uniform distribution randomly distributes the operations across the keyspace. Zipfian distribution makes a few keys extremely popular whereas the majority are chosen very seldom, according to the zipfian curve. Latest distribution is like the zipfian distribution, except that the most popular records are the most recent ones. Finally, multinomial distribution associates different probabilities to different operations to achieve different ratios of reads to writes. Different workloads in the core package are defined by varying the operations and their distribution in the keyspace.

4. Driver Design

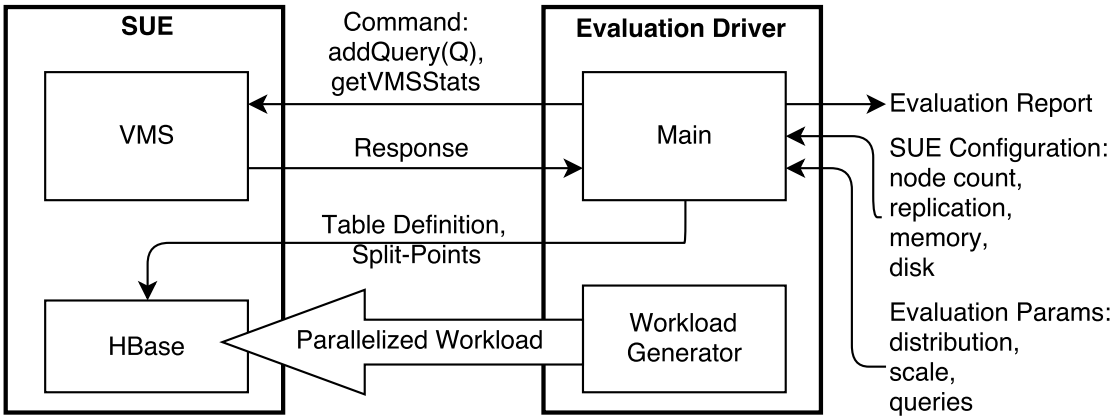


Figure 4.1.: Evaluation Overview

In this chapter we design the framework for the evaluation of the View Maintenance System. We begin by formally defining the System Under Evaluation(SUE), the Evaluation Driver, and the workload. We next look at the first step of the evaluation which is base table creation using the SplitPointsGenerator. Next, we extend our discussion on MapReduce Parallelization from the background material and see how MapReduce can be exploited to generate the workload. Finally, we put it all together and see how the evaluation works.

4.1. Delineating the System Under Evaluation

The System Under Evaluation(SUE) consists of an instantiation of the VMS running on top of HBase. We looked at the detailed design of the VMS in the background material in section 2.4. Here we abstract away the details and look at the different components of the SUE from the standpoint of the evaluation driver(see figure 4.2). The HBase layer of the SUE contains the base tables and the view tables. These tables are split across multiple RegionServers(RS). The base tables are created by the evaluation driver using HBase's client API. The view tables are created by the VMS Master component upon receiving a request from the evaluation driver. The base tables receive updates via the client API. These updates are propagated to the RegionServer(RS) component of the VMS. It delegates processing of these updates to a ViewManager(VM) component which in turn updates the corresponding view tables. The VMS Master is also responsible for gathering and consolidating statistics reports from the view managers and reporting

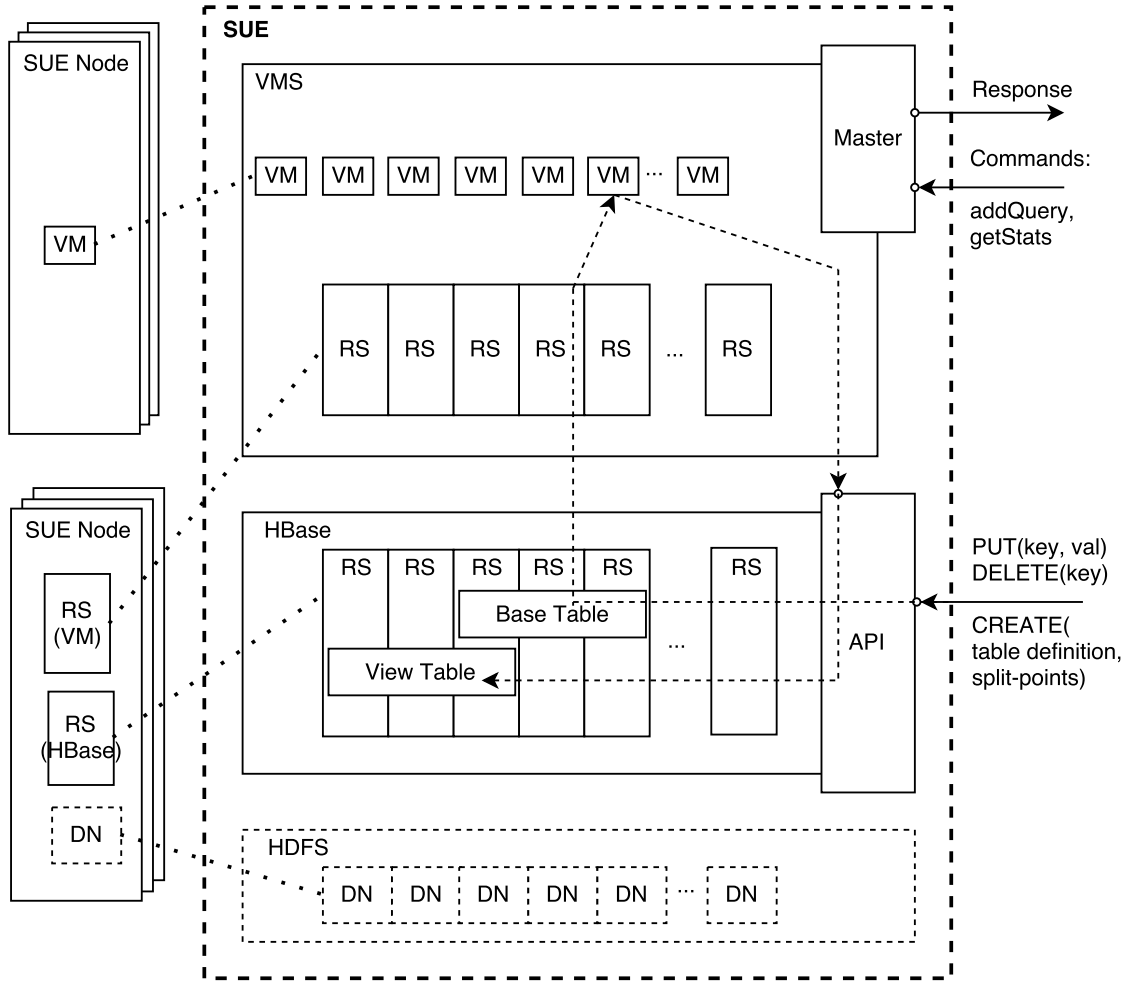


Figure 4.2.: System Under Evaluation (SUE)

them when requested. Thus we see that there are two points of interaction with the SUE - one at the API exposed by HBase and another at the VMS Master(see figure 4.1). We also note that HBase itself runs on top of HDFS. This aspect of the SUE is transparent to the evaluation driver as long as it is not collocated on the HDFS data nodes(DN) hosting HBase data and is thus prevented from interfering in the performance of HBase.

4.2. Workload

Workload(or load) consists of insertion of data into HBase base tables. The base tables follow the TPC-H database schema and so the data is generated using TPC-H's DBGen utility. Data is generated in parts as flat-files. These are parsed and pushed to HBase using its PUT client API. The data can be generated at different scales(as specified by TPC-H). This allows us to load the SUE at different scales depending on available system resources. Data can also be inserted at different levels of concurrency to simulate

different number of parallel users. The order in which the generated data is inserted can be used to control the load distribution across the SUE. Inserts spread-out across the key-space lead to a uniform load across HBase RSs and corresponding VMS RSs. Inserts made sequentially across contiguous sections of the key-space lead to a hot-spotted workload which moves across the key-range from one end to the other.

4.3. Evaluation Driver

The Evaluation-Driver is responsible for applying the workload on the SUE and for receiving performance statistics from the VMS Master in return. It interacts with the SUE at two points - the HBase API and the VMS-Master, using its HBase Client and VMS-Client components respectively. The statistics contain througputs and latencies for individual view types as well as for the VMS system as a whole. The driver is also responsible for creating pre-split TPC-H base tables. It generates split points for each table based on the data scale factor and SUE configuration(nodes, memory, replication). The workload is generated in a distributed fashion using Hadoop jobs, where each job is responsible for loading one table. The LocalShellService allows each mapper to generate a part of the table data and push it to HBase using the HBaseClient. Workload concurrency and distribution is varied using different mapper configurations.

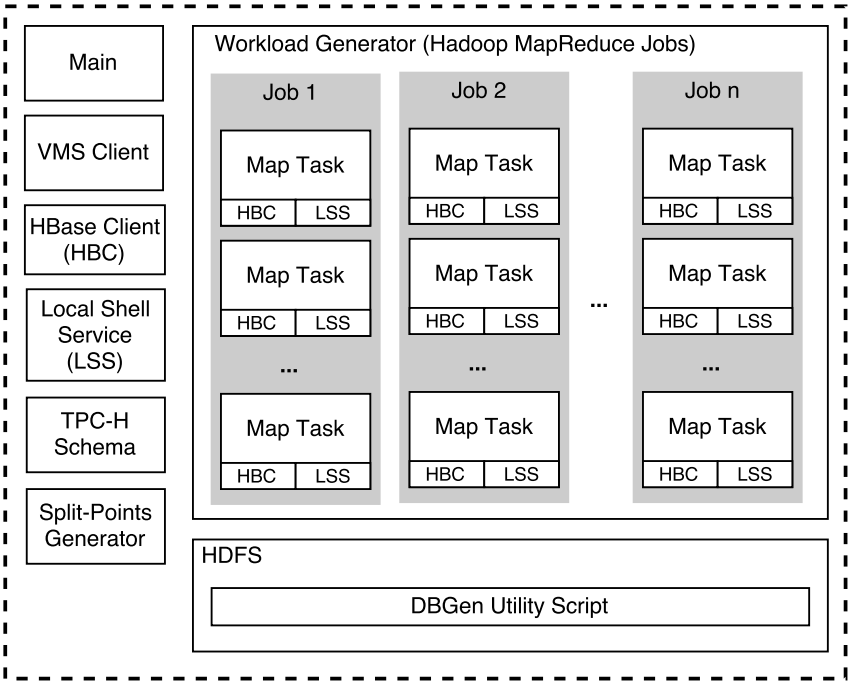


Figure 4.3.: Evaluation Driver

4.4. Base Tables Pre-Splitting: SplitPointsGenerator

When a table is first created, HBase, by default, will allocate only one region for the table. This means that initially, all requests go to a single region server, regardless of the number of region servers or the distribution of the workload. To prevent this it is important that the base tables are pre-split. This avoids costly splitting as the table starts to fill up, and ensures that the table starts out already distributed across many servers. The HBase client API allows specifying a set of split-points in the keyspace of a given table during creation. HBase will then split the regions based on these points. This works for us because the key ranges of the different tables in TPC-H are known beforehand for a given scale factor. We can therefore find the split points depending on how many regions we want to pre-split a table into.

The SplitPointsGenerator component of the driver finds split points for each table by finding the optimal number of regions the table should be split into. Too few regions means a sub-optimal load distribution. On the other hand, region count is upper bounded by the region server memory size. Each region has an associated in-memory memstore (per column family) and the total memstore usage can not go beyond a certain fraction of the memory. This total memstore usage fraction is configurable and has a default value of 40%. Furthermore, more regions means more work for the master, which will have to spend a lot of time assigning and moving them around for load balancing.

The number of regions per RS(for one table) is therefore predicted as follows [20]:

$$\text{region count} = \frac{\text{(RS memory) * (total memstore fraction)}}{\text{(memstore size) * (\# column families)}}$$

Base table information like table size and key range is hardcoded into SplitPointsGenerator for the scale factor of 1 (see table). It then takes the scale factor, HBase memstore size, and memstore usage fraction as input to generate split points using the above formula. These split points are used for initial base table creation.

4.5. Workload Generation using MapReduce

4.5.1. Exploiting Task Level Concurrency

HBase CompleteBulkLoad

The use of MapReduce jobs to generate the workload is motivated from the HBase CompleteBulkLoad utility[20]. This utility is usually used together with the HBase ImportTsv utility[20] to perform initial database load into HBase. The data is put to HDFS in the form of flat files containing tab separated values for each HBase row. The ImportTsv utility takes these flat files and converts them to HBase store files(HFiles) using the table definition and split points(if any). It then stores these HFiles back to

HDFS. The CompleteBulkLoad utility called LoadIncrementalFiles then takes these HFiles and loads them into HBase. Thus the output from ImportTsv(the Hfiles) serves as input to the LoadIncrementalFiles. It is important to note that the HFiles are already stored on HDFS and so LoadIncrementalFiles has to only tell the region servers where to find them. There is no movement of HFiles in the second step of incremental load. Also, it is important to note that the incremental load bypasses the WAL altogether and directly adds HFiles to HBase. Since both these utils are implemented as MapReduce jobs the HFile generation and incremental load both benefit from parallelization.

To bulk load TPC-H data into our evaluation database the TPC-H part files would serve as input to ImportTsv. However, since the WAL is bypassed no corresponding load would be seen on the VMS RegionServers and so it is not useful for our purpose of evaluating the VMS. At the same time, the mechanics of the HBase bulk load make it amply clear that MapReduce can be used to generate a distributed load for HBase and the overlying VMS system. We now explore this in detail.

Workload Setup

A MapReduce job is configured for each base table that we want to load. An input file is created based on the number parts into which we want to break the total table data. Each row of the input file consists of a part number as key and total parts as value. This tells the mapper which processes the row which table part it is responsible for generating and pushing to HBase. The mapper generates the table part using TPC-H's DBGen script which is added to MapReduce's distributed cache beforehand. It then parses the part file and creates the corresponding HBase put statements which it then submits to HBase via its client API. Many mappers work in parallel thus simulating multiple concurrent users. Furthermore, the order in which the parts are listed in the input file decides the sequence in which data is pushed to HBase. This in turn decides the load on the HBase and corresponding VMS RegionServers. If part files are processed in order(e.g. part 1/9, part 2/9, part 3/9, ...) then only one Region Server(responsible for current key range) is loaded - this gives a *hot-spotted* workload. If on the other hand, part files are generated at uniformly spaced intervals(e.g. part 1/9, part 4/9, part 7/9, ...) then the workload is *uniform*.

Uniform Workload

The first step is configuring the number of mappers. This decides how many concurrent insert streams will be present in the system. For a uniform workload the input file lists part numbers sequentially as seen in figure 4.4. Now since the number of mappers is explicitly configured, the input file is split into the number of mappers instead of by size. This ensures that each mapper has one part of the file to work on. Each mapper works on its part sequentially and so the different mappers are at any time working on parts which are uniformly spread out. Thus each mapper generates load for a different Region Server. The load is spread out.

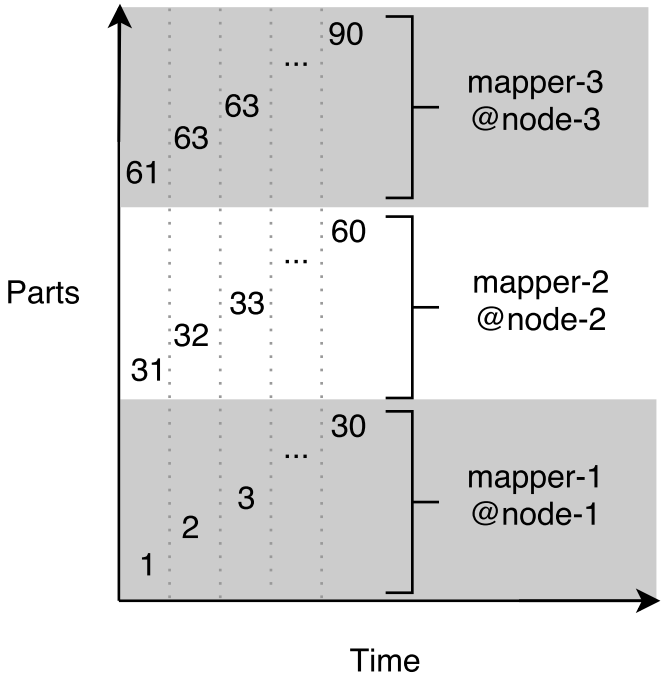


Figure 4.4.: Uniform Workload Parts Distribution

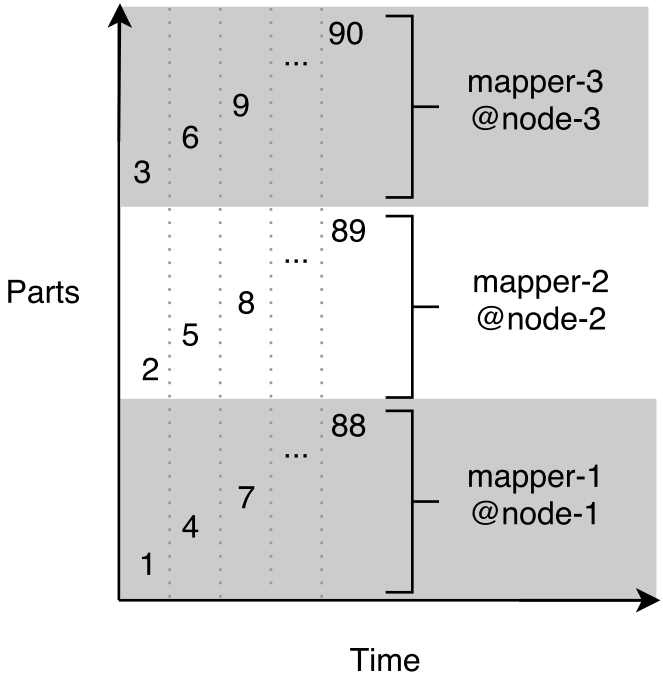


Figure 4.5.: Hot Spotted Workload Parts Distribution

Hot-Spotted Workload

In case of a hot-spotted workload, the parts are listed in the input file as shown in figure 4.5. Each mapper works on an adjacent part. Thus at any given time all mappers are generating inserts from a particular range of keys. This sends the load to a single(or possibly few) Region Servers thus hot-spotting the workload.

Thus we see how the input file allows us to control the distribution of the workload across the RS. In case of the spread-out workload the number of map tasks helps us control the granularity of the split. The number of map tasks per node and the number of nodes which together define how many map tasks are active at any given time, help us control the degree of concurrency of the workload for both spread-out and hot-spotted requests. This also suggests that these different parameters can also be further customized to achieve a variety of other workloads. However this is not explored further in this thesis.

4.5.2. Exploiting Job Level Concurrency

One MapReduce job is configured for each base table. This results in 8 different jobs which which are run concurrently. Now the number of map tasks is proportional to the number of regions into which a base table is pre-split. This means a larger base table writes to HBase over a larger bandwidth using many more map tasks compared to a smaller table. This could starve jobs writing to smaller base tables. We get around this by making use of MapReduce FairScheduler and its *sizebasedweight* property. It takes into account job sizes in calculating their weights for fair sharing. By default, weights are only based on job priorities. Setting this flag to true will make them based on the size of the job (number of tasks needed) as well,though not linearly (the weight will be proportional to the log of the number of tasks needed). This lets larger jobs get larger fair shares while still providing enough of a share to small jobs to let them finish fast.

4.6. Putting It All Together

The basic evaluation framework is visualized in figures 4.6 and 4.7. The former visualizes a uniform workload whereas the later shows a hot-spotted workload. The input file can be modified in a variety of other ways to achieve more custom workloads, but this is left for future work. Furthermore, it must be noted that we consider the workload for a single base table only. Similar workloads are applied on all base tables as part of parallel jobs. These jobs are configured using a *FairShareScheduler* as discussed before. This ensures that all tables are loaded at a rate proportionate to their sizes, while at the same time larger tables are not starved in favor of finishing smaller loads first. We now look at a simplified single base-table evaluation scenario.

The SUE consists of one base table split into 90 regions spread across 3 HBase region servers(RS). Each RS is collocated with a VMS Region Server which is responsible for reading any updates to the HBase RS's WAL and passing it on to the VMS View Managers(VM). Each VMS RS has 3 VMs registered with it. These are assigned to a

hash-ring over the View Table keyspace so that updates are distributed equally to the VMs using consistent hashing[6]. The RSs store the regions in the form of HFiles on corresponding HDFS data nodes. But these are transparent to our evaluation purposes.

The Evaluation Driver divides the input workload into 90 parts, each corresponding to one region in HBase. This allows us better control and visualization of the load distribution. 3 Map Tasks are configured for the Job and this causes the input file to be split into 3 parts, each serving as input for one mapper. Each mapper works sequentially on the key-value items in their part. Each item is picked up and the key and value are split up to get the partNumber and totalParts respectively. This information is given to the LocalShellService which invokes the DBGen script to obtain the corresponding TPC-H part file. This file consists of rows of the table, which is being loaded, as comma separated values. The HBase Client parses the part file and converts it into HBase client API's PUT statements. It then submits these PUTs to the SUE. As can be seen from the following figures the workload distribution changes with the order of the key-value items in the input file. In the case of uniform distribution each mapper submits PUT statements to a different HBase RS. In case of hot-spotted distribution all mappers submit their PUT statements to the same RS.

This simplified scenario can be extended to include multiple mappers per driver node. This would be logically equivalent to the simplified scenario with the only difference being smaller part files and a heavier (more parallelized) workload. Furthermore, as the scale of the evaluation is increased, more map tasks can be added to the job to account for a larger input data set. However the map tasks can not be increased indefinitely and at some point new nodes will need to be added to the driver once the existing nodes become network saturated. At smaller workloads the VMS performance is expected to be bound by the number of map tasks. This is because a small number of updates can be processed by the VMs as soon as they come in. At higher workloads the true test of performance begins. For different scales and distributions the performance is measured against different number of VMs for each different view type. We will see the variation evaluation parameters in detail in the upcoming section on Evaluation and Results.

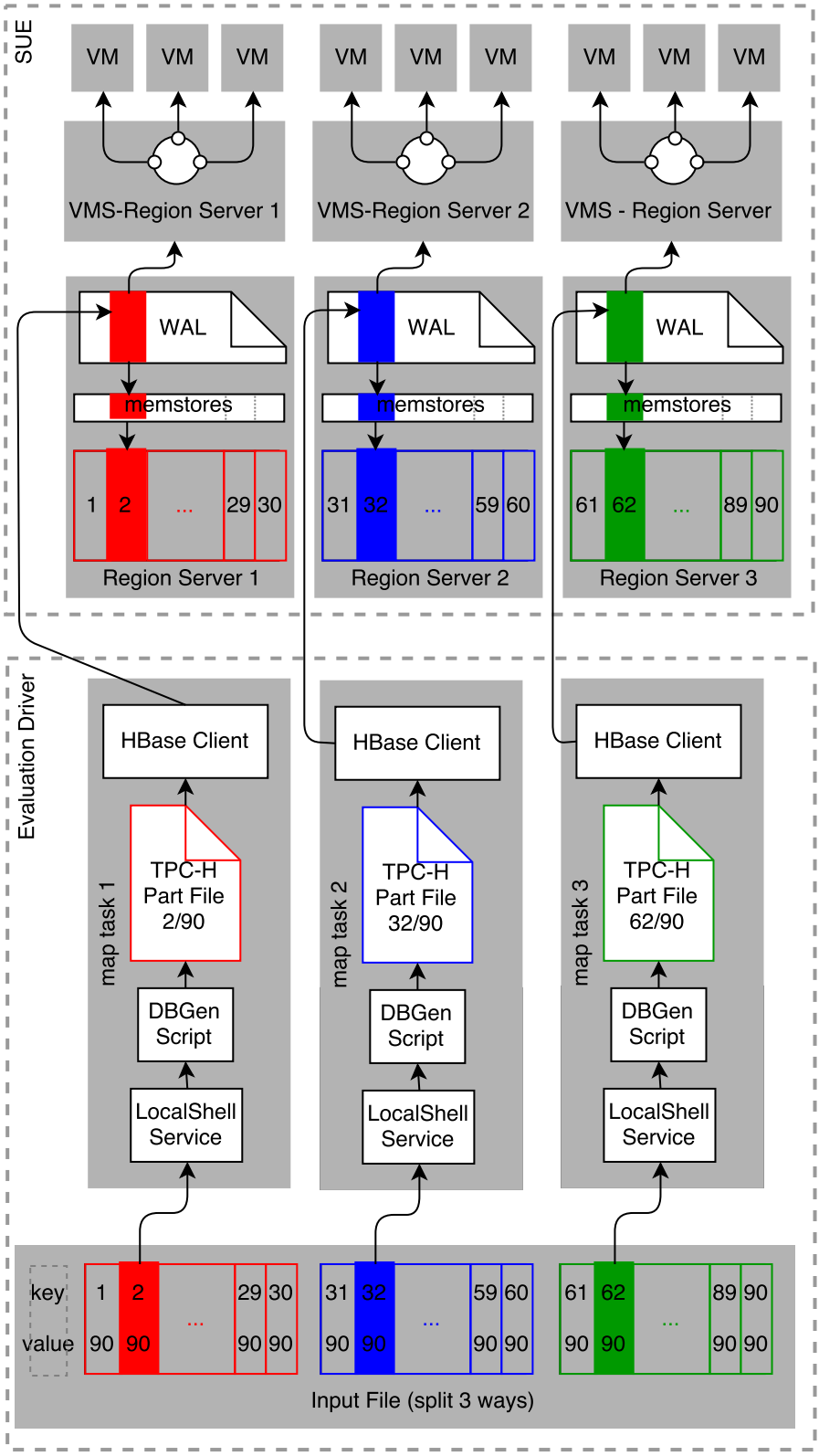


Figure 4.6.: Evaluation - Uniform Workload Distribution

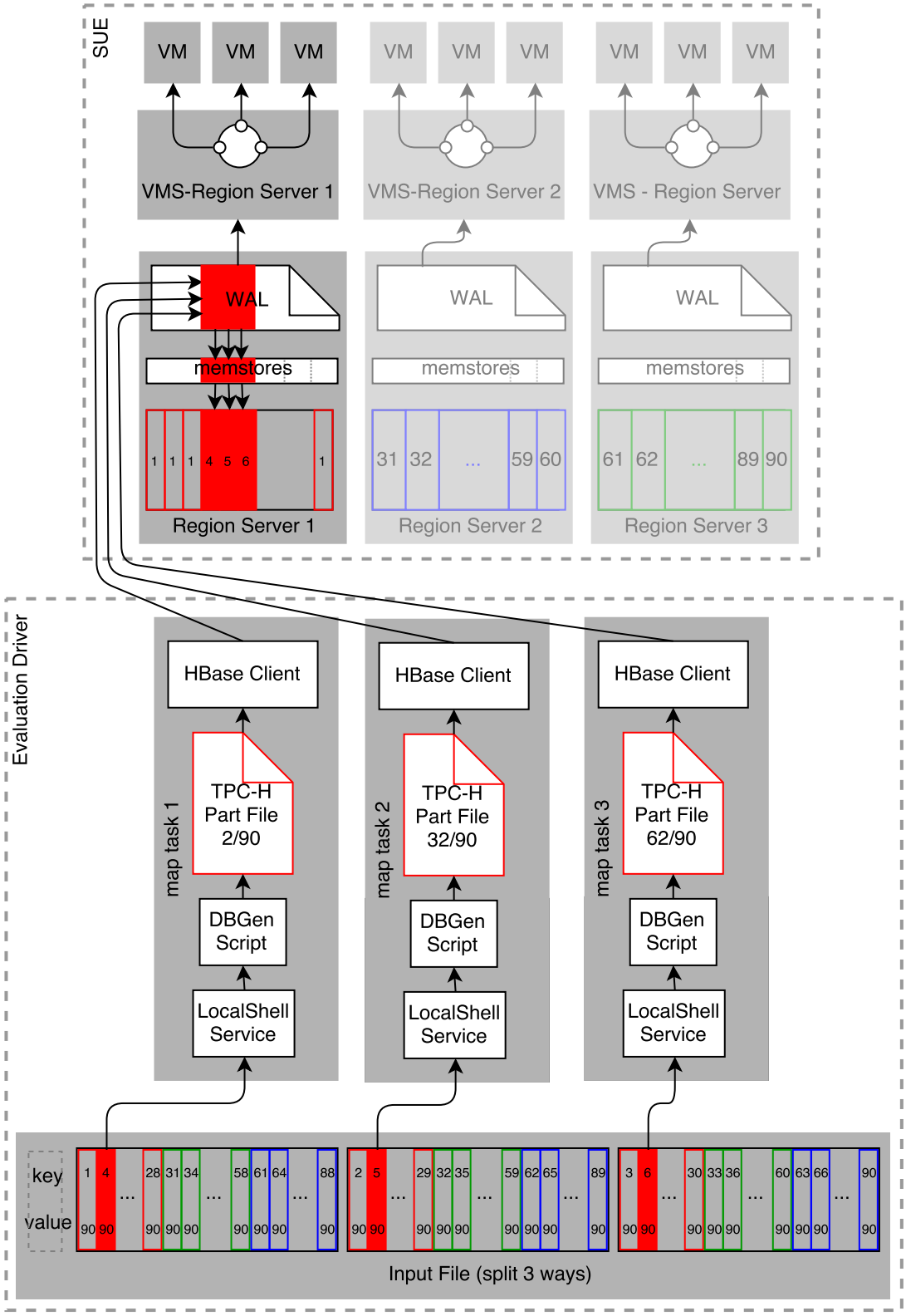


Figure 4.7.: Evaluation - Hot-Spotted Workload Distribution

5. Driver Implementation

Deployment
Config
Class or component diagrams
show small code snippets
This is followed by a look at a few more components of the driver - namely the VMSClient, the HBase Client, the LocalShellService, and the BarChartGenerator.

This is followed by an examination of the TPC-H DBGen utility and how it can be utilized to generate the workload data.

5.1. VMSClient

The VMSClient is the first of two points of interaction between the Evaluation Driver and the SUE. It communicates with the VMSMaster over TCP by submitting commands to one port and receiving responses from another. It uses the addQuery(Q) command to ask the VMSMaster to configure a view corresponding to query Q. It waits for an acknowledgement from the VMSMaster without which it retries for a fixed numvber of times. It also uses the getThroughput(detailed/summary) command to ask for the performance statistics of the VMS system. The detailed option allows for performance statistics to be received for each VMS RS individually. The summary option gives a consolidated system wide statistics report. The main statistics reported are throughput and latency, both overall and for individual auxiliary view types.

5.2. HBaseClient

The HBaseClient is the second interaction point between the Evaluation Driver and the SUE. It communicates to HBase using its client API. Its first task is to submit base table creation statements to HBase. It makes use of the SplitPointsGenerator discussed earlier to add split points to the create statements for pre-splitting the base tables. Its second task is to facilitate insert statements from each map task and as such it forms a part of every mapper.

5.3. LocalShellService

An important aspect of our workload generation is the creation of part files locally at each mapper node. These files are created by invoking the TPC-H DBGen scripts which are copied to each node as part of th e job setup phase(via MapReduce's *distributed*

cache mechanism). Each of these files is deleted as soon as the task is over and its data has been insertedin into HBase. The LocalShellService which forms part of each map task is instrumental running scripts loclally for the creation and deletion of these part files.

5.4. ChartGenerator

The ChartGenerator is responsible for visualizing the final evaluation report based on the performance statistics received from the VMSMaster. It makes use of the JFreeChart library. The plots are primarily between workload vs througput or latency for different view types. As such, one of the important tasks of the ChartGenerator is to parse the performance statistics received from the VMSMaster and convert them to a format suitable for the charting library.

5.5. TPC-H DBGen Utility

The TPC-H DBGen utility script is used to generate the workload data. The data is generated in the form of flat text files which can be parsed and converted into HBase PUT statements. An important aspect of the utility is that the data can be generated in parts. This allows a MapReduce job to split data generation across multiple map tasks and over a period of time. The utility accepts data scale factor, table name, part number, and total parts as parameters.

```
$./dbgen -s <scaleFactor> -S <partNumber> -C <totalParts> -T <tableName>
```

It is important to note that a part file is the unit of granularity of our workload. A part file contains a contiguous set of table rows. This means all the inserts generated from one part file go to the same HBase RS and subsequently generate a load in the same VMS RS. Thus controlling the sequence in which part files are generated allows us to control the load distribution. The number of part files which are generated in parallel allows us to control the number of users the workload simulates. Both these aspects are controlled using the MapReduce job configuration responsible for a particular table as we will see in section 4.5. The DBGen script files are stored in HDFS and transferred to the distributed cache of each job beforehand. The distributed cache ensures that the script is copied to each map node before job execution begins. This transfer is a one time activity for all nodes and forms part of the job setup time.

6. Evaluation and Results

Params desc Workload desc General process Describe Each experiment general observation and interpretation Fewer plots but good quality Vary 1 parameter each experiment. Never change together. So use variations of same experiment.

Split factor Replication factor Number of RSs Number VMs

Time taken to insert workload as parallelization increases. Install nodes script

6.1. Experiment 1 - Validating Workload Distribution

6.1.1. Uniform Load Distribution

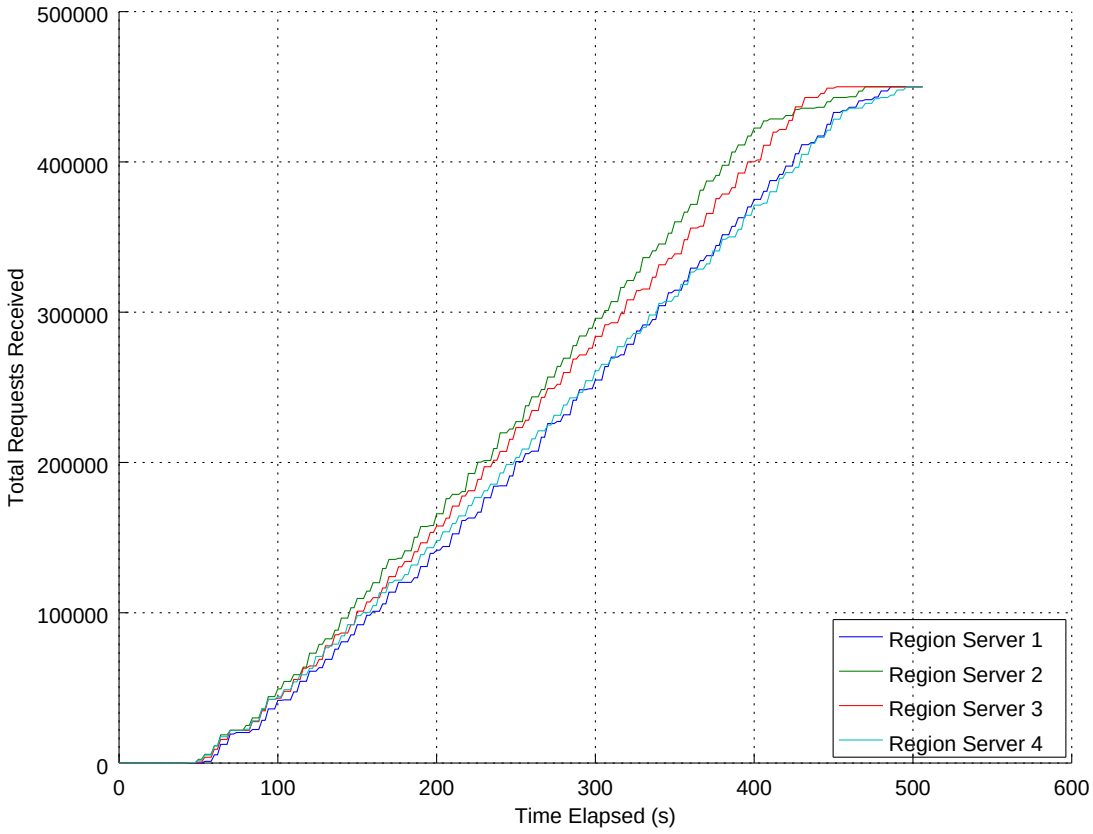


Figure 6.1.: Uniform Workload Progress

6.1.2. Hotspotted Load Distribution

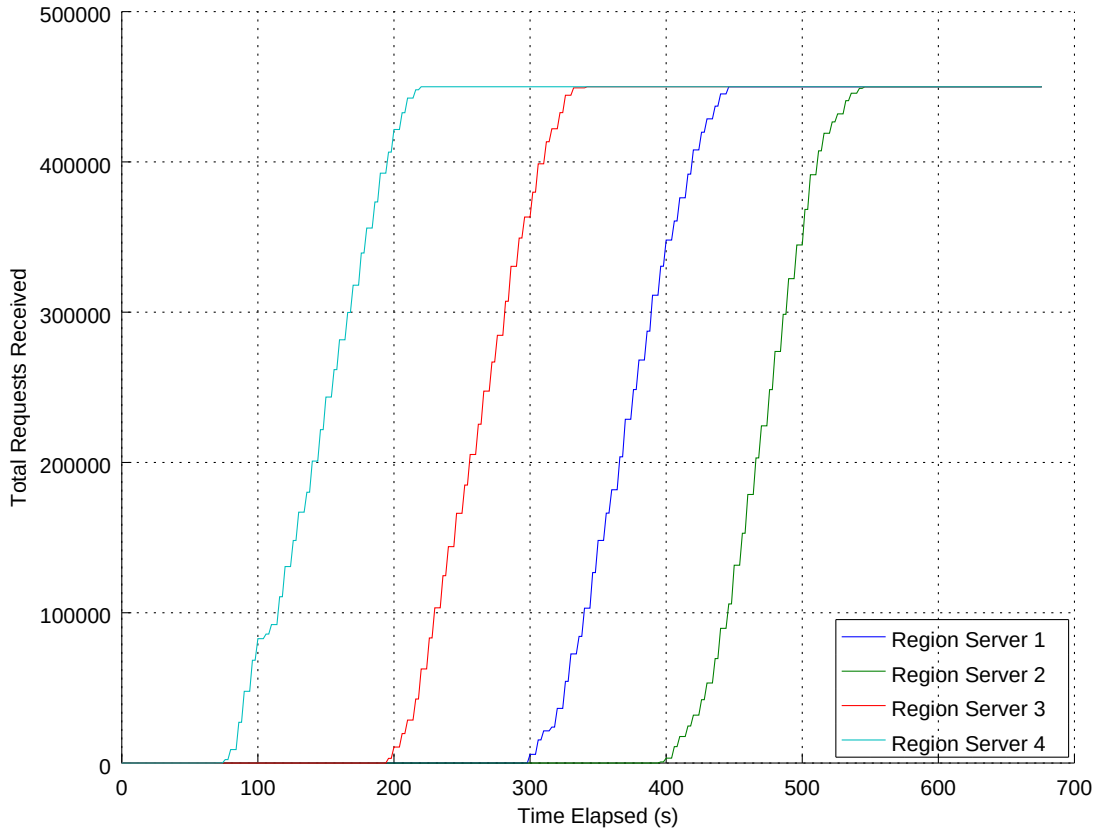


Figure 6.2.: Hotspotted Workload Progress

6.2. Experiment 2 - Evaluating Workload Parallelization

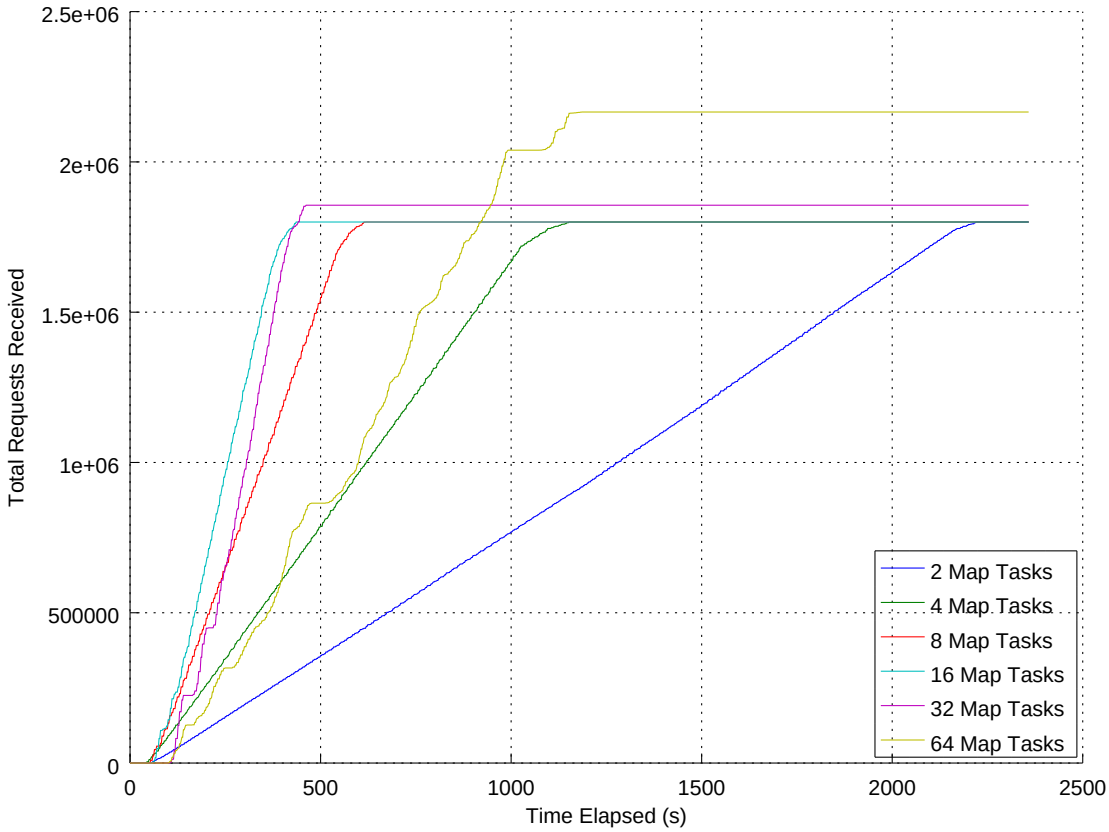


Figure 6.3.: Progress at various degrees of parallelization

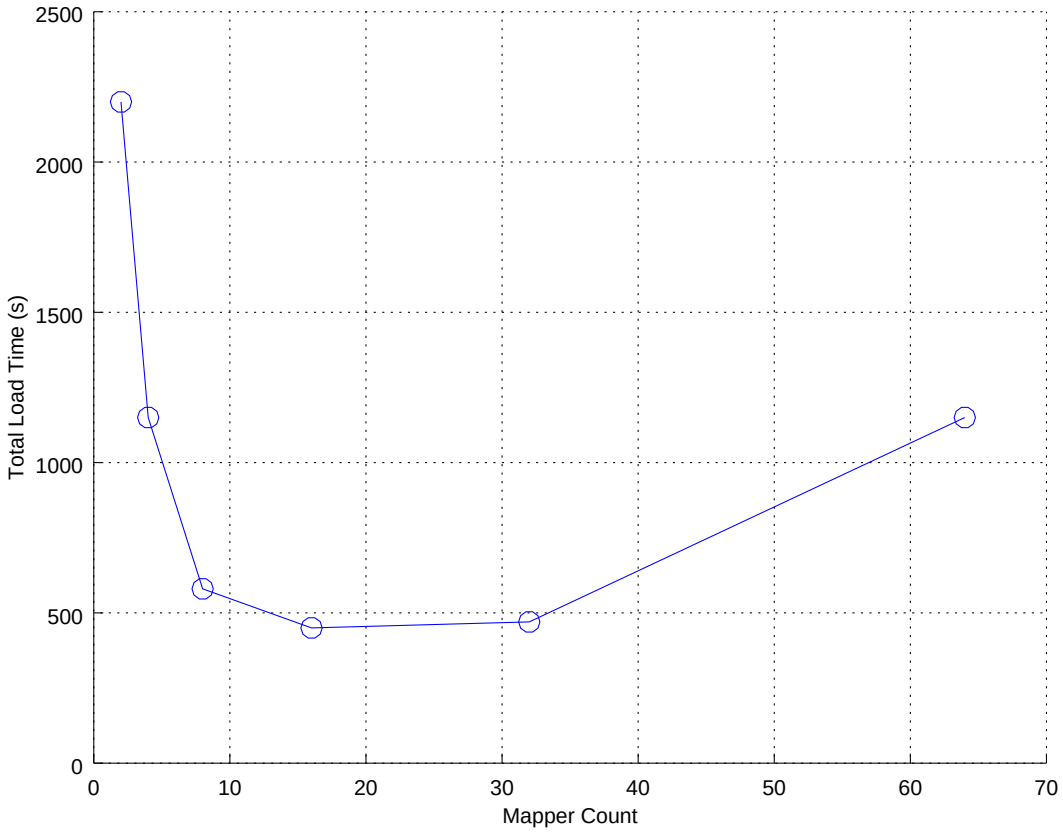


Figure 6.4.: Load times at various degrees of parallelization

6.3. Experiment 3 - Evaluating Workload Granularity

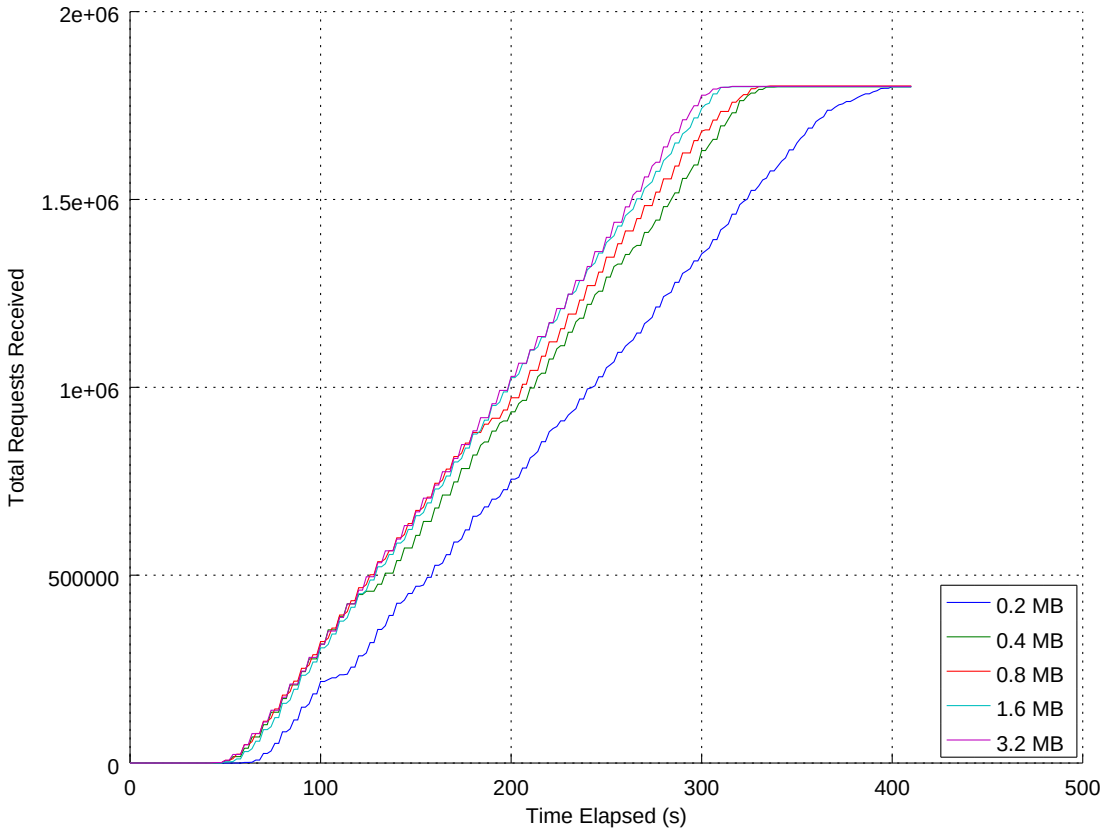


Figure 6.5.: Progress at various task sizes

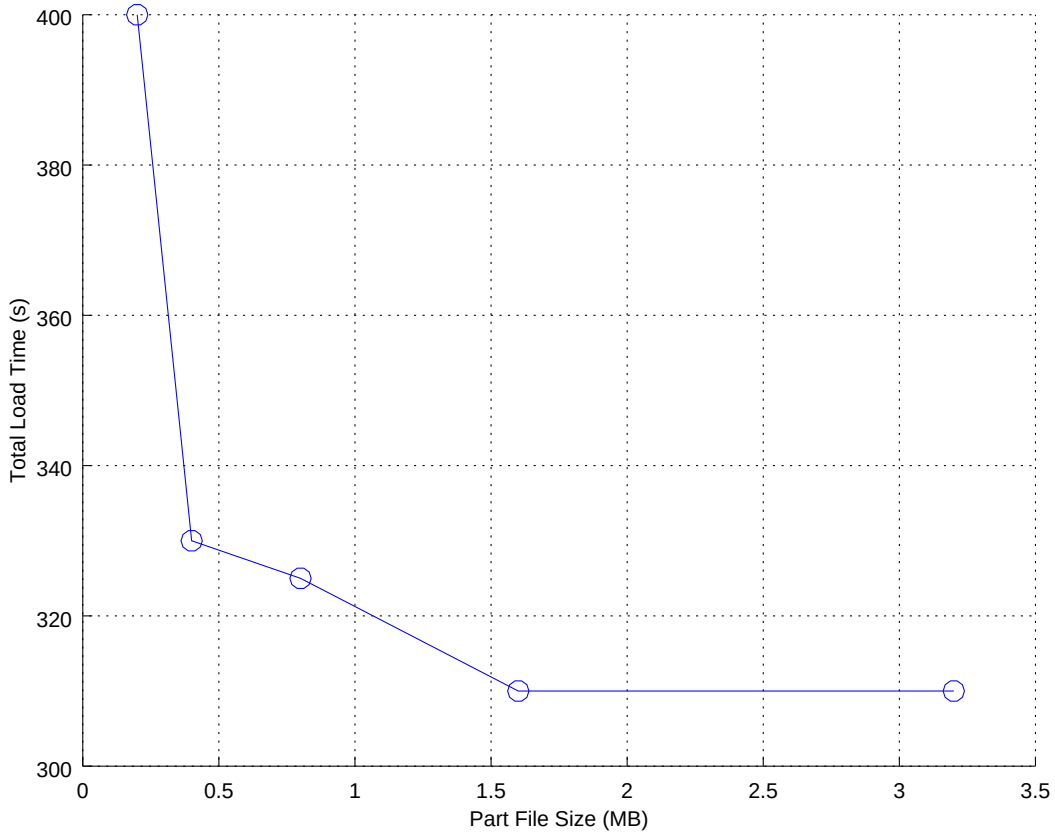


Figure 6.6.: Load times at various task sizes

7. Conclusion

8. Future Work

refer YCSB 7.1 Tier 3—Availability refer YCSB 7.2 Tier 4—Replication

Appendices

A. TPC-H

A.1. TPC-H Table Schema

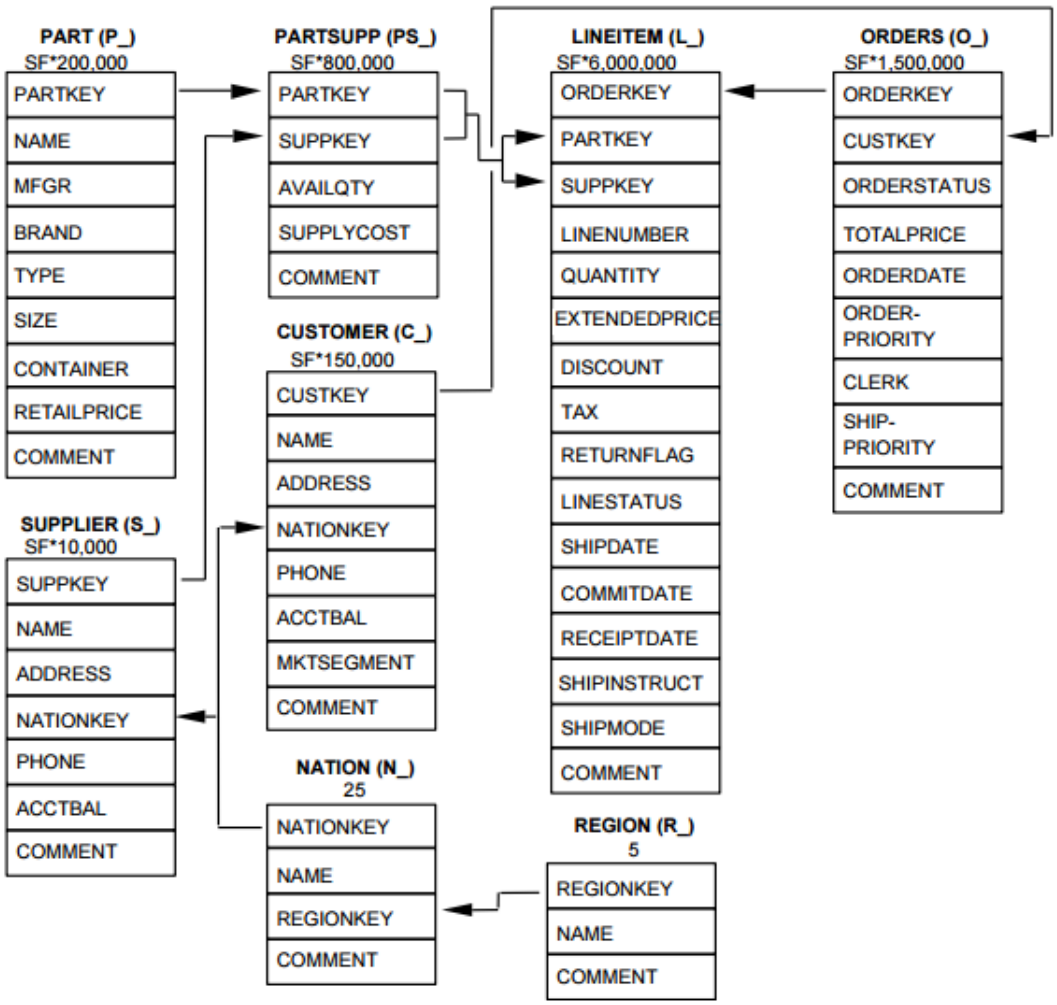


Figure A.1.: TPC-H Database Schema[7]

A.2. TPC-H Queries

A.3. TPC-H Refresh Functions

A.3.1. New Sales Refresh Function (RF1)[7]

This refresh function adds new sales information to the database. It inserts new rows into the *orders* and *lineitem* tables in the database following the scaling and data generation methods used to populate the database. The set of rows to be inserted must be produced by DBGen using the -U option. This option will produce as many sets of rows as required for use in multi-stream tests.

RF1 Definition:

LOOP (SF * 1500) TIMES
INSERT a new row into the *orders* table
LOOP RANDOM(1, 7) TIMES
INSERT a new row into the *lineitem* table
END LOOP
END LOOP

A.3.2. Old Sales Refresh Function (RF2)[7]

This refresh function removes old sales information from the database. It removes rows from the *orders* and *lineitem* tables in the database to emulate the removal of stale or obsolete information. The primary key values for the set of rows to be deleted must be produced by DBGen using the -U option. This option will produce as many sets of primary keys as required for use in multi-stream throughput tests. The rows being deleted begin with the first row of each of the two targeted tables.

RF2 Definition:

LOOP (SF * 1500) TIMES
DELETE FROM *orders* WHERE *o_orderkey* = [value]
DELETE FROM *lineitem* WHERE *l_orderkey* = [value]
END LOOP

A.4. TPC-H Metrics

A.4.1. Processing Power Metric (*Power@Size*)

$$Power@Size = \frac{3600}{\sqrt[24]{\prod_{i=1}^{22} QI(i,0) \times \prod_{j=1}^{2} RI(j,0)}} \times SF$$

QI(i,0) = Execution time for the ith query in seconds
RI(j,0) = Execution time for the jth refresh function in seconds
SF = Scale factor

A.4.2. Throughput Power Metric (*Throughput@Size*)

$$Throughput@Size = \frac{S \times 22}{T_s} \times 3600 \times SF$$

S = Number of streams
T_s = Total execution time for the throughput test in seconds
SF = Scale factor

A.4.3. Composite Query-Per-Hour Metric (*QphH@Size@Size*)

$$QphH@Size = \sqrt{Power@Size \times Throughput@Size}$$

A.4.4. Price/Performance Metric (*Price-per-QphH@Size*)

$$\text{Price-per-QphH@Size} = \frac{\$}{QphH@Size}$$

Dollar = Dollar price of the system under testing

Bibliography

[1] A. Homer. (). Materialized view pattern. [Accessed: 30-November-2016], [Online]. Available: https://msdn.microsoft.com/en-us/library/dn589782.aspx.

[2] R. Chirkova and J. Yang, "Materialized views," *Foundations and Trends® in Databases*, vol. 4, no. 4, pp. 295–405, 2012, ISSN: 1931-7883. DOI: 10.1561/1900000020.

[3] (). Apache phoenix | views. [Accessed: 30-November-2016], [Online]. Available: https://phoenix.apache.org/views.html.

[4] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, "Pnuts: Yahoo!'s hosted data serving platform," *Proc. VLDB Endow.*, vol. 1, no. 2, pp. 1277–1288, Aug. 2008, ISSN: 2150-8097. DOI: 10.14778/1454159.1454167.

[5] T. Rabl and H.-A. Jacobsen, "Materialized views in cassandra," in *Proceedings of 24th Annual International Conference on Computer Science and Software Engineering*, ser. CASCON '14, Markham, Ontario, Canada: IBM Corp., 2014, pp. 351–354.

[6] J. Adler, M. Jergler, and A. Jacobsen, "Dynamic scalable view maintenance in kv-stores."

[7] *Tpc benchmark(tm) h standard specification*, 2.17.1, Transaction Processing Performance Council (TPC), Nov. 2014.

[8] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, ser. MSST '10, Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–10, ISBN: 978-1-4244-7152-2. DOI: 10.1109/MSST.2010.5496972.

[9] L. George, *HBase: The Definitive Guide, 2nd Edition*. O'Reilly Media, Inc., 2017.

[10] A. Thanopoulou, P. Carreira, and H. Galhardas, "Benchmarking with tpc-h on off-the-shelf hardware," *ICEIS (1)*, pp. 205–208, 2012.

[11] *Tpc benchmark(tm) ds standard specification*, 2.3.0, Transaction Processing Performance Council (TPC), Aug. 2016.

[12] (). What's new about the tpc-ds 2.0 big data benchmark. [Accessed: 30-November-2016], [Online]. Available: http://data-informed.com/whats-new-about-tpc-ds-2_0-big-data-benchmark/.

[13] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC '10, Indianapolis, Indiana, USA: ACM, 2010, pp. 143–154, ISBN: 978-1-4503-0036-0. DOI: 10.1145/1807128.1807152.

[14] G. Kamat. (). Ycsb, the open standard for nosql benchmarking, joins cloudera labs. [Accessed: 30-November-2016], [Online]. Available: http://blog.cloudera.com/blog/2015/08/ycsb-the-open-standard-for-nosql-benchmarking-joins-cloudera-labs/.

[15] A. Ghazal, T. Rabl, M. Hu, F. Raab, M. Poess, A. Crolotte, and H.-A. Jacobsen, "Bigbench: Towards an industry standard benchmark for big data analytics," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '13, New York, New York, USA: ACM, 2013, pp. 1197–1208, ISBN: 978-1-4503-2037-5. DOI: 10.1145/2463676.2463712.

[16] B. D Gowda and N. Ravi. (). Bigbench: Toward an industry-standard benchmark for big data analytics. [Accessed: 30-November-2016], [Online]. Available: http://blog.cloudera.com/blog/2014/11/bigbench-toward-an-industry-standard-benchmark-for-big-data-analytics/.

[17] I. Corporation. (). Big bench workload development. [Accessed: 30-November-2016], [Online]. Available: https://github.com/intel-hadoop/Big-Data-Benchmark-for-Big-Bench.

[18] T. Rabl, A. Ghazal, M. Hu, A. Crolotte, F. Raab, M. Poess, and H.-A. Jacobsen, "Bigbench specification v0.1," in *Revised Selected Papers of the First Workshop on Specifying Big Data Benchmarks - Volume 8163*, New York, NY, USA: Springer-Verlag New York, Inc., 2014, pp. 164–201, ISBN: 978-3-642-53973-2. DOI: 10.1007/978-3-642-53974-9_14.

[19] T. White, *Hadoop: The Definitive Guide, 4nd Edition*. O'Reilly Media, Inc., 2015.

[20] *The Apache HBase™ Reference Guide*. Apache Software Foundation, 2015.