ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Systems@ETH

Master's Thesis

Daedalus
A Distributed Crescando System

by
Georgios Giannikis

November 1st, 2008 - April 30th, 2009

Supervisor:
Philipp Unterbrunner
Advisor:
Prof. Gustavo Alonso

Systems Group

ETH Zurich
Department of Computer Science
8092 Zurich
Switzerland

Abstract

This thesis deals with the design and implementation of Daedalus, a platform for creating and maintaining distributed materialized views based on the Crescando Storage Engine. The Crescando Storage Engine in a main-memory storage engine that offers predictable response times and throughput on unpredictable workload, however the supported dataset size is limited by the size of main memory. To overcome this limitation, a distributed architecture is proposed, where every node stores a subset of the data such that the latency predictability property is maintained.

This thesis first investigates different dataset partitioning scenarios, based on knowledge of the expected workload. An optimal partitioning scenario reduces the number of nodes involved in answering a single query, for most of the expected queries. To simplify the overall system architecture, only static partitioning scenarios are investigated.

Next, the thesis deals with an implementation of a query router, called "aggregator" in Daedalus terminology. The query router has full knowledge of all crescando storage nodes and the range of their respective datasets. The purpose of the aggregator is to forward incoming queries to the nodes that can answer them. Additionally, the aggregator is responsible for merging results from different nodes, in the case where a query can only be answered by more than one node.

The query router serves as the communication channel between the distributed system and the clients. It must offer high availability by being replicated and stateless. To allow a query router to be stateless, we can take advantage of the fact that there is a single writer to our system (since it is a materialized view of some table). The single writer is expected to persistently store its state and to not experience permanent failures.

The last issue that this thesis deals with, is the implementation of a client-side library to connect and query the materialized view. The client-side library is schema independent, so that any modification of the dataset schema does not require updates at the client side.

Preface

This thesis is a part of the global distributed dictionary research project of the Systems Group at the Department of Computer Science, ETH Zürich. It is submitted for the partial fulfillment of the Master Program (MSc ETH) in Computer Science. The duration of the thesis is 6 months, from November 1st, 2008 to April 30th, 2009 in the Informations and Communications Systems Group under the supervision of Philipp Unterbrunner and Prof. Gustavo Alonso.

Acknowledgements

First of all, I would like to thank my supervisor Philipp Unterbrunner for offering me this challenge during my Master Thesis. I am very grateful for his quick responses and solutions provided during difficult parts of the project. I would also like to thank my mentor, Prof. Gustavo Alonso, for his support during the whole Master Program at the ETH Zürich. Last but not least, I would like to thank Prof Donald Kossmann for the advices that were given at all levels of the research project. Eventually, this master thesis would not have been completed without the invaluable support of my family.

Contents

Chapter 1

Introduction

1.1 Motivation

Industries such as the financial, insurance, or the travel industry today face considerable technical challenges as their database systems need to grow simultaneously in multiple dimensions. This is because the use of service oriented architectures at an enterprise scale and, with it, the increasing number of services to be provided result not only in increased throughput requirements, but also in higher query diversity as well as increased data freshness and access latency requirements.

At the heart of the problem lie fact tables of the core business: financial transactions, shopping baskets, travel bookings etc. These form the entry point of a highly diverse set of use cases, covering the spectrum from on-line transaction processing (OLTP) to data warehousing (DW). Particularly troublesome in this respect, and the focus of our work, are analytic queries with stringent data freshness constraints, i.e. queries which cannot be delegated to a stale, off-line replica of the data (i.e. a DW system).

The traditional solution for such queries are redundant materialized views and secondary indexes. Unfortunately, views and indexes are limited in their ability to scale in multiple dimensions for two main reasons. First, indexes are by nature a hotspot in any database system and thus an impediment to linear scale-up on modern multi-core platforms under mixed (read-write) workloads. Second and more importantly, the set of views and indexes effectively define the set of queries that can be answered by a system in reasonable time and with reasonable overhead, making it hard to scale in the dimension of query diversity

While creating a new index or materialized view may seem trivial, in any real system there will always be a point at which materializing yet another view or creating yet another index is no longer economical. That is because views and indexes are expensive, both in terms of hardware resources for storage and updating, and human resources for maintenance.

As companies evolve and data centers grow, it becomes increasingly difficult to meet service level agreements (particularly response times) and data freshness constraints for all services and corresponding queries. What is sought for is a simple, predictable, and highly scalable answer to growth in *all* dimensions, including update throughput and query diversity. The goal is an affordable system able to answer *any* query on-line, with bound latency (say, two seconds), on *live* data under a heavy update load.

To this end, we propose a radical simplification. Instead of using multiple materialized views and indexes, build a single, de-normalized, unindexed view of the fact table. All queries and updates are executed by way of continuous, massively parallel scans in main memory. Scanning the data sidesteps the view selection and index maintenance problems. It offers predictable throughput and latency under an unpredictable workload. Simply speaking, it takes query diversity and read-write contention out of the equation, reducing multi-dimensional growth to a throughput problem, which can be solved by adding hardware alone.

Since simplicity and scalability are in vain if the required amount of hardware is not economical, the first step of our work was to push the performance limits of a single node built of commercial-of-the-shelf (COTS) hardware. We already addressed this issue in Unterbrunner et al [7], where a multi-threaded storage engine, the "Crescando Storage Engine", is proposed as a replacement to traditional solutions. In this respect, it is clear that scans will never outperform an index under a key-matching, highly selective, read-only workload. However, we identified and implemented a number of key optimizations, which, in combination, allow the Crescando Storage Engine to predictably meet latency and freshness requirements as low as a few seconds, while offering very high throughput even for mixed workloads.

The Crescando Storage Engine being a single machine main memory storage imposes new limitations in our system. At the time this report was written, the main memory a COTS machine could support was at most 64 GBytes. Additionally, even though the Crescando Storage Engine performance scales in the number of CPU cores that are present, current hardware limits the number of CPU cores per machine to just 32. In order to overcome these new issues, we propose a simple distributed system, *Daedalus*. Daedalus is built on top of the existing Crescando Storage Engine, offering the same guarantees and service level agreements.

Daedalus sets no limit on the dataset size it can store, as more nodes running the Crescando Storage Engine can be added in the distributed system. These "Crescando Storage Nodes" store only a portion of the total dataset. Furthermore, Daedalus performance scales in the number of participating Crescando Storage Nodes.

We will demonstrate by means of a thorough performance evaluation that Daedalus not only meets the throughput, latency, and data freshness requirements of the travel industry, it is also significantly more robust to concurrent updates and unanticipated query predicates than traditional solutions.

1.2 Case Study: Amadeus

Amadeus IT Group is a world-leading service provider for managing travel-related bookings (flights, hotels, rental cars, etc.). Its core service is the Global Distribution System (GDS), an electronic marketplace that forms the backbone of the travel industry. The world's largest airline carriers and many thousand travel agencies use the GDS to integrate their flight booking data. This work is part of a larger effort that aims to replace the current mainframe and traditional database infrastructure with more scalable and cost-efficient alternatives based on clusters of multi-core commercial off-the-shelf (COTS) machines.

The core database in the Amadeus GDS contains dozens of millions of flight bookings. For historical and performance reasons, the authoritative copy of each booking is stored in a denormalized BLOB (binary large object) of a few kilobytes, directly accessible through a unique key. For the bookings that need to be kept on-line, this results in a single, flat fact table of several hundred gigabytes in size. This BLOB table currently sustains a workload of several hundred updates and several thousand key-value look-ups per second.

Key-value access works well enough for all transactional workloads faced by the system. However, it is ill-suited to answer the increasing amount of real-time, decision-support queries that select on non-key attributes, for example: "give the number of disabled persons in first class, who depart from Tokyo to a destination in the US tomorrow." Queries like this are increasingly common and feature stringent latency constraints, because operational decisions are made based on their results.

The traditional way to support such queries is to create more indexes in the relation. Such indexes allow faster execution of more diverse queries, however in case of updates a large number of indexes would require locking of all index structures, which deteriorates performance.

To support such queries, Amadeus maintains a growing number of materialized relational views on the BLOB table, some of which are updated in real-time through a proprietary event streaming architecture. The very existence of these materialized views implies that there are few joins in the workload. The vast majority of queries are of the form `SELECT <Attr1>, <Attr2> ... FROM <View> WHERE ...`, with occasional aggregation.

The largest existing view is a denormalization of flight bookings: one record for every person on a plane. This is the view used for the performance evaluation in this paper, and we will refer to it as *Ticket* in this context. A Ticket record is approximately 350 bytes in size (fixed), and consists of 47 attributes, many of which are flags with high selectivity (e.g., seat class, handicapped, vegetarian). Since one travel booking may be related to multiple persons and flights, Ticket contains hundreds of millions of such records.

Ticket is updated a few hundred times per second, in real-time. Update rates may be many times higher for brief periods, as bad weather or security incidents

can cause large bursts of passenger reaccomodation requests. The update load is increasing at a lower rate than the query load, but is already causing severe problems with regard to index maintenance in the current setup.

The view is used in a large number of data services: from generating the passenger list of a single flight to analyzing the customer profile of different airlines and markets (pairs of <source, destination> airports). Since the system has reached a level of complexity where adding views and indexes is no longer feasible let alone economical, a growing number of queries on Ticket do not match the primary index on <flight number, departure date>.

As a result, more and more queries have to be answered in batch (off-line) using full-table scans, with a dramatic impact on performance during this period. Other queries which do not match the index and do not allow for batch-processing are simply not allowed. As a solution to all these queries that do not warrant a view of their own, we propose a single instance of Ticket implemented as a Daedalus table.

To make matters worse, Ticket - being a materialized view of a base table - has to reflect all updates of the base table. Creating redundant instances of the materialized view or adding more indexes on it, creates more problems, as the update rate increases. Currently, the Ticket view receives an average of 1 updates per gigabyte per second, however the peak update rate is much higher. To make matters worse, Amadeus expects the update rate to increase in the future. The size of the materialized view is around 300 GB and is not expected to radically increase in the future. The Ticket tables stores bookings in a range of two years: a year in the past and a year in the future.

1.2.1 Requirements

We summarize the requirements of our use case as follows:

- **Query Latency** Any query must be answered within two seconds.
- **Data Freshness** Any update/insert/delete must be applied and made visible within two seconds.
- **Query Diversity** The system must support any query, regardless of its selection predicates.
- **Update Load** The average load is 1 update/GB*sec. Peak load is 20 updates/GB*sec for up to 30 seconds.
- **Scalability** The system must scale linearly with the read workload by adding machines (scale-out) and cores to individual machines (scale-up).

Our design goal is to maximize read (i.e. select query) throughput per core under those requirements.

1.3 Outline of the Thesis

Most of the effort during this thesis was spent on designing a simple Clock Scan-based distributed system. Such a task requires extensive analysis of our use case and the algorithms used. This document is organized as follows. In Chapter 2 we present the Crescando Storage Engine, which provides predictable latency for any workload. Chapter 3 uses the Crescando Storage Engine as the basic block of a distributed architecture. Chapter 4 presents an analysis on our use case's workload which allows us to make decisions on how the distribution of dataset should be performed. Chapter 5 provides all the details of our implemented distributed system. In Chapter 6 we analyze the performance of the Crescando distributed system under a realistic workload. Finally, Chapter 7 concludes the report with a summary of our main insights and an outlook on future work.

Chapter 2

The Crescando Storage Engine

2.1 Overview

The core of Crescando is a novel collaborative-scan algorithm, called Clock Scan. The Clock Scan algorithm batches incoming queries and models query/update processing as a join between the queries and update statements on the one side and the table on the other side. Instead of indexing the data of the table, Clock Scan creates temporary indexes of the queries and updates that are currently being evaluated. For main-memory databases, this setup is particularly effective, since random access is cheap.

In Clock Scan, an operation is either a *query* (unnested, SQL-style `SELECT` statement) or an *update*. In this thesis, we use the term update for any unnested, SQL-style `INSERT`, `UPDATE`, or `DELETE` statement. We write `UPDATE`, `INSERT`, or `DELETE` whenever we want to make a distinction.

The main advantage of Clock Scan is that it can sustain any workload while having predictable latency. In addition, Clock Scan supports write monotonicity and serializability.

The Crescando Storage Engine was introduced and analyzed in a previous paper by the author [7]. Here we only provide an overview insofar as it aids the understanding of the limitations and possibilities of our distributed system. We will start our description of the Crescando Storage Engine in a bottom-up fashion, starting from memory segments.

2.1.1 Memory Segment

The Crescando Storage Engine partitions data in what is called *memory segments*. A memory segment is assigned to one dedicated processor core that runs the Clock Scan algorithm. The size of the memory segment is constant during runtime, however it is a configurable parameter of the engine and in fact a very crucial one, as we will see.

Every memory segment maintains queues of running operations, a queue of pending operations and a queue of produced results. The interface of the memory segment consists of two functions: enqueue an operation and dequeue a result. Rather than enqueuing an operation and waiting for the result, the upper layer is expected to *pipeline* a large number of concurrent operations in the segment.

The Clock Scan algorithm scans the memory segment by wrapping around it, is resemblance to the pointer of a clock. At the beginning of the segment, Clock Scan tries to move operations from the pending operations queue into the running operations queue and then will continue scanning for tuples matching any operation in the running operations queue. Any tuples matching the operations are copied to the results queue. The time required for a full scan to be completed is of great importance, since it determines the time that is required to answer a query.

Prior to actually scanning for tuples, Clock Scan attempts to partition the set of running operations, based on the predicates they contain. The partitioning of the operations is performed by examining the predicated attributes and some statistical information that allow selectivity estimation. Every partition is implemented as a tree index for ranged predicates or as a hash index for single value predicates. Simple speaking, this reduces the complexity of looking into a partition from $O(N)$ (checking every operation) to $O(\log N)$ or $O(1)$. While scanning, Clock Scan performs query/update-data joins over sets of queries/updates. Once the scan reaches the end of the segment, these temporary query indexes are discarded and the running operations are considered complete and are removed.

The maximum size of the incoming operation and outgoing results queues are also configurable parameters of the engine. If a small number of operations is allowed to run simultaneously, then all of them can fit in L1 cache, allowing faster evaluation of the operations. However, allowing more operations to run concurrently, the gain from the query indexes results in sub-linear time complexity. Finally, a big number of concurrent operations creates too large indexes that do not fit in the processor cache, resulting into workload thrashing. We will further discuss the effect of the queue sizes in section 2.3.

Finally, as an optimization, hard affinity is used to bind a memory segment to a processor. This reduces any task switching since scan threads can no longer migrate between processors. Furthermore, a memory segment is allocated in memory local to the bound processor core by using *libnuma*, a NUMA API for Linux [5]. NUMA allocated memory allows us to deploy a number of memory segments on a single multi-core system while preventing contention on the memory bus.

2.1.2 Many-core system

Since recent hardware systems involve an increasing number of processor cores, it would be a waste of resources to use only a single memory segment

per machine. As already described, memory segments are assigned to a single processor core. Deploying more than one memory segment, requires management and organization that is provided by a software module that is called the "Controller".

The Controller manages and pipelines operations into memory segments. It is also responsible for receiving results from memory segments, combining them and outputting a result to the user of the engine. The Controller also maintains a queue of incoming operations as well a queue of outgoing results, tuples that matched the operations. Upon receiving an operation in the incoming queue, the Controller decides, based on a predefined *Segmentation Strategy*, which memory segments are responsible for evaluating the given operation.

Segmentation Strategy defines the way data is partitioned in memory segments. The naïve approach is to select a memory segment in a round-robin fashion to store a tuple (for `INSERT` operations). In this case, in event of a query or update, all memory segments have to be involved in evaluating it. Of course, round robin creates additional load, but also it represents the worst case of this multi-core system.

A different approach is to have data replicated on every memory segment. The Controller could distribute queries in a ROWA (read-one-write-all) manner, favoring query execution. However in this approach, the number of records that can be stored in the system is very low.

The most promising scenario to distribute tuples in memory segments, is to select a tuple attribute and use its hash value to calculate which memory segment should execute the given operation. An operation that includes an equality predicate on the hashed attribute will be evaluated by only one memory segment, while all others will be evaluated by every single memory segment, which is the worst case as already mentioned. A careful choice of the hash attribute is important as it reduces the computational cost of the matching queries and updates. The primary key of the table is a natural candidate. The advantage of the Crescando Storage Engine compared to traditional index based systems is that, even if a query or update does not select on the partitioning attribute, it will be answered in roughly the same amount of time.

The interface of the Controller is as simple as the one of memory segments, providing exactly the same functions. Once a Controller receives an operation, it calculates which memory segments should evaluate this operation based on its Segmentation Strategy. Then it tries to enqueue the operation into these memory segments. When a result is dequeued from some memory segment, it is placed in the outgoing results queue. For this purpose the Controller has to keep track of which operations are currently being evaluated and which have finished. When all memory segments have finished evaluating a specific operation, the Controller marks the operation finished, by creating a special *end-of-operation* result.

2.1.3 Recovery

Since the Crescando Storage Engine is an implementation of a main-memory table, there is a need for a mechanism that maintains durable replicas of the dataset. One might argue that Crescando, being targeted as an implementation of materialized views, does not require durability on any level. However, the Crescando Storage Engine provides this functionality, since rebuilding a large, business critical view might be unfeasibly slow and/or have an intolerable performance impact on the base tables.

In the Crescando Storage Engine durability is supported by means of checkpointing and logical logging of the updates. For checkpoints, a background thread is responsible for periodically issuing unpredicated queries (queries that match any tuple). The results of these queries are serialized and stored in a flat file (the checkpoint) on a persistent storage (hard disk). Considering logging, every update must be appended to the log prior to being confirmed (write-ahead logging). A background thread periodically (e.g. every second) flushes the log to disk and truncates it to the timestamp of the last checkpoint.

Of course the existence of the recovery mechanism causes some interference with the normal execution of Clock Scan on the memory segments. This is mostly caused by the vast number of results that have to be generated and passed from one thread to another. The effects of recovery are studied in Section 2.3.

2.2 Evaluation

The system mentioned so far has been implemented as a shared library for 64bit POSIX systems, written in C++ with a few lines of inline assembly. Similar to other embedded and main-memory databases, a small fraction of schema-dependent code is generated by a schema compiler and loaded at runtime. The engine offers a simple C interface with two main functions: enqueue an operation, and dequeue a result.

The current implementation includes a number of different segmentation policies. In this section, we only evaluate the round-robin and the hash partitioning policies. Additionally, a number of different algorithms for query indexing, reordering and creating joins with data are available. A throughout analysis of these algorithms has been conducted in [6]. We will only evaluate our system using the Index Union Join algorithm, since it promises the highest performance.

2.2.1 Test Platform

We conducted all experiments on a 16-way machine built from 4 quad-core AMD Opteron 8354 (“Barcelona”) processors with 32 GB of DDR2 667 RAM. Each core has 2.2 GHz clock frequency, 64 KB + 64 KB (data + instruction) L1 cache, and 512 KB L2 cache. According to AMD, this setup has a cumulative memory bandwidth of over 42 GB/sec [2]. The machine was running a 64-bit Linux SMP kernel, version 2.6.27.

Algorithm 1: Attribute Pick Algorithm

```
Input: N, D, s
Output: P
Z ← Zipf(s, N);                          //initialize Z as Zipf distribution
V ∼ B(N, 1/D);                           //get random V acc. to binomial dist. B
for v = 1 to V do
    a ∼ Z ;                              //get random a according to Z
    p ← Z[a];                            //get probability p of a according to Z
    Z[a] ← 0 ;                           //remove a from Z
    Z ← Z/(1 − p) ;                      //re-normalize remaining Z
    P ← P ∪ a ;                          //add a to result set P
```

2.2.2 Workload Generation

We recreated the Ticket schema introduced in Section 1.2, and ran all experiments on Amadeus' live data. For the experiments we generated two types of workload. The first was a representative subset of the real-world queries of Amadeus with their relative frequencies, including inserts, updates, and deletes. The second was a synthetic workload with variable predicate attribute skew.

Amadeus Workload

At the time of writing, the average query executed against Amadeus' Ticket table has 8.5 distinct predicate attributes and fetches 27 of the 47 attributes in the schema. 99.5% of the queries feature an equality predicate on <flight number, departure date>.

`UPDATE`s occur at an average rate of 1 update/GB*sec, typically affect just a few records, and always feature equality predicates on booking number and arrival time. As discussed in Section 1.2, update rates may be many times higher for brief periods. `INSERT`s and `DELETE`s occur at about 1/10th the rate of `UPDATE`s. `DELETE`s typically affect specific records just like `UPDATE`s, but occasionally purge large numbers of old tuples based on date predicates.

Synthetic Workload

As motivated in the introduction, we were not as much interested in the performance under the current, index-aware workload, as in the performance under future, increasingly diverse and unpredictable workloads; i.e., we were interested in workloads where users ask *any query they want*, uninhibited by knowledge of the table's physical organization.

For this purpose, we created a synthetic workload as follows. We kept all the updates of the Amadeus workload (their composition is fairly stable over time), and replaced the real queries by a synthetic set of queries with a particular, *skewed* predicate attribute distribution. To create such a skewed set of predicate attributes for a synthetic query, we invoked Algorithm 1.

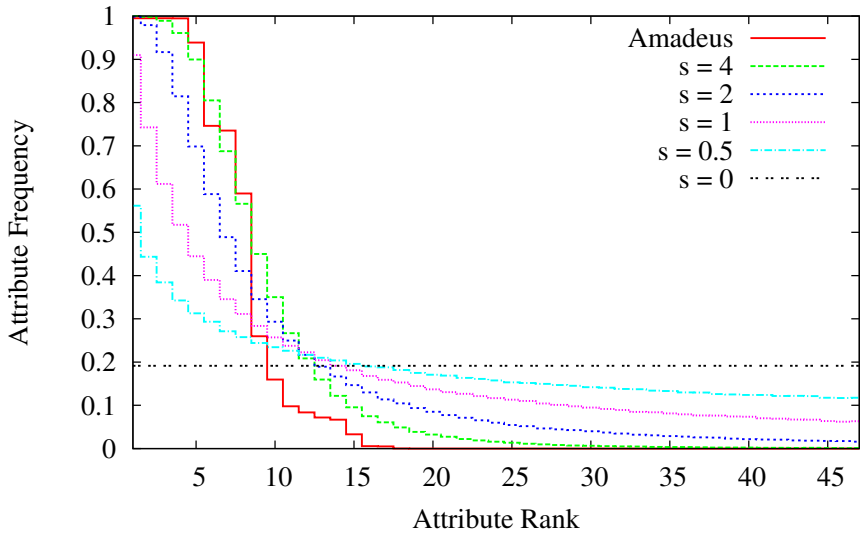


Figure 2.1: **Synthetic Workload Chart**
N = 47, D = 0, Vary s

Executing the algorithm with $N = 47$ (total attributes in schema), $D = 9$ (average predicates per query), and variable shape parameter s yields the attribute frequencies given in Figure 2.1. s is the characteristic exponent of the Zipf distribution used by Algorithm 1 to initialize the probability function Z. The figure includes the attribute frequencies of the current Amadeus workload for comparison. Notice that the area under each synthetic curve is $D = 9$, as desired. The area under Amadeus' curve is roughly 8.5.

2.2.3 Experimental Results

In order to simulate our case study, we used a fragment of Amadeus Ticket data. Since the data provided by Amadeus was only 2GB, we used a data generator that created records based on the Amadeus data. This way, we generated a dataset of 15 GB of ticket data, which was hash segmented by flight number. In all our experiments we reserved one processor core for our benchmark driver, allowing us to use up to 15 memory segments.

Metrics

All experiments focus on two performance metrics: query/update throughput and query/update latency. Throughput refers to the number of queries/updates that were executed and a valid result was returned to the benchmark driver. Operations that were still unanswered when the experiments finished were considered as failed. By latency we refer to the time from a query being generated by the benchmark driver, to the benchmark driver receiving the full result. For latency measurements, we always provide the 50th, 90th, and 99th percentile.

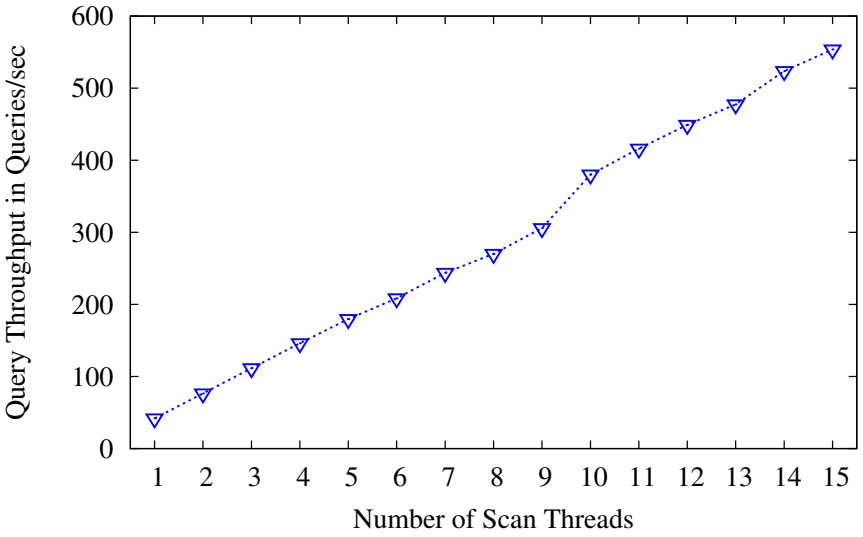


Figure 2.2: **Multi-core Scale-up: Query Throughput**
Amadeus Read-Only, RR Part., Vary Scan Threads

Multi-core Scale-up

The first experiment is focused on how the Crescando Storage Engine scales on modern multi-core platforms. For this purpose, we saturated the system with a read-only variant of the Amadeus workload. We used *round-robin* partitioning, which, as already mentioned, means that every scan thread had to process every incoming operation. We kept the dataset size constant at 15 GB, while increasing the number of memory segments. This translates into decreasing memory segment size for increasing numbers of scan threads.

Clock Scan scales from 42.3 to 558.5 queries/sec as seen on figure 2.2. The linearity of the curve also demonstrates that Clock Scan is *CPU bound* on modern NUMA machines. The bump in throughput between 9 and 10 threads is due to the fact that for 10 threads the segments have become small enough to be statically allocated on the local NUMA node, giving maximum NUMA locality. Note that we limited the running queue sizes to 512 operations. This gave a median query latency of around 1 second. Having more queries share a cursor would yield somewhat higher peak throughput at the expense of additional latency.

Query Latency

As already mentioned in Chapter 1.2, our targeted use case has strict requirements in terms of latency. In the introduction we explained why traditional approaches fail to guarantee service level agreements when the workload contains updates or queries that do not match the materialized view's index. The following three experiments focus on how the Clock Scan algorithm behaves in term of latency for a variable workload.

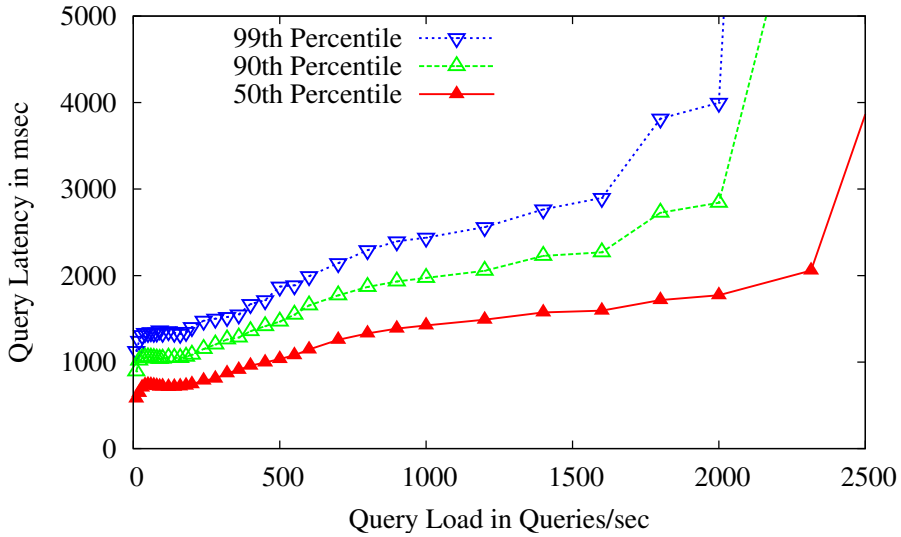


Figure 2.3: **Robustness to Variable Workload: Query Latency**
Amadeus Read-Only, Hash Part., Vary Queries/sec

Variable Query Load

As a starting point, we investigated the robustness of Crescando to bursts in query volume. For this purpose, an increasing number of queries resembling the Amadeus' workload was created as we were measuring the 50th, 90th, and 99th percentile of latency. Figure 2.3 shows the results. For this experiment we used hash segmentation over the memory segments, which explains why the engine is able to sustain higher throughput than in the previous experiment. Also, note that we did not bound the incoming and outgoing queue sizes in order to identify the point at which the Crescando Storage Engine starts thrashing the load.

As for latency, one can see that it is very much constant up to about 200 queries/sec. We found that this is the point where the working set of Clock Scan (indexes and less-selective queries) exceed the L1 cache. Between 200 and about 2,000 queries/sec, latency is logarithmic in the number of queries. Beyond 2,000 queries/sec, latency increases sharply. We ran a number of micro-benchmarks and found that 2,000 queries/sec is the point where the working set exceeds the L2 cache. At about 2500 queries/sec, the system fails to sustain the load and input queues grow faster than queries can be answered.

Query Latency: Variable Update Load

Our second latency-related experiment focuses on the behavior of Clock Scan under a mixed (queries and updates) workload. We created a constant 1,000 Amadeus queries per second and gradually increased the update load to the point where the system could not sustain it (2,300 updates/sec), while we were measuring the 50th, 90th, and 99th percentile of query latency.

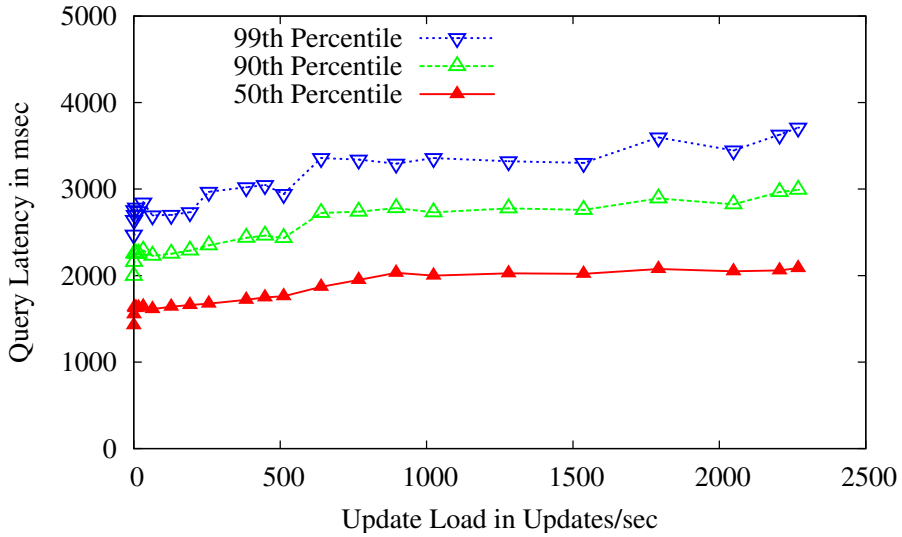


Figure 2.4: **Robustness to Variable Workload: Query Latency**
Amadeus Mixed, Hash Part., 1,000 Qu./sec, Vary Upd./sec

The results, depicted in Figure 2.4, show that query latency increases by about 35% between 0 and 1,000 updates/sec. Beyond this point, latency hardly grows further. Notice that query latency does not increase sharply right before Crescando fails to sustain the query load. This is because Clock Scan maintains separate pending operation queues for queries and updates.

We conclude that Clock Scan is a very update-friendly algorithm, especially since the hash partitioning on flight number only helps with the *query load*. Updates do not have a matching predicate. Instead, most of them have an equality predicate on booking number, on which the current, traditional solution maintains a secondary index.

Query Latency: Variable Query Diversity

The Crescando Storage Engine is designed to answer any query at any time with predictable latency and impact on the system. This implies a certain robustness to variance in query predicates. To put this property to the test, we measured the sensitivity of throughput and latency to synthetic read-only workloads with varying parameter s (*cf.* Figure 2.1).

To keep latency within reason for low values of s, we limited the operations queue sizes to 1024 queries. The resulting query latencies are shown in Figure 2.5, while the achieved query throughput is shown in Figure 2.6. Latency stays in the region required by our use case (*cf.* Section 1.2) for s up to 1.5. Beyond this point, latency increases quickly. To put things into perspective: $s = 0$ represents an (unrealistic) uniform distribution of predicate attributes. The workload changes radically between $s = 1.5$ and $s = 0.5$ as Figure 2.1 indicates, so the latency increase is more the result of a "workload cliff" than a

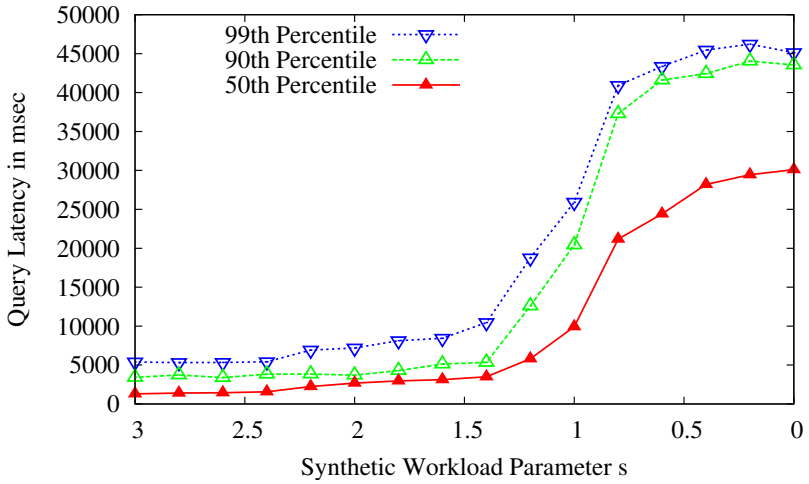


Figure 2.5: **Robustness to Variable Workload: Query Latency**
Synthetic Read-Only, Hash Part., Vary s

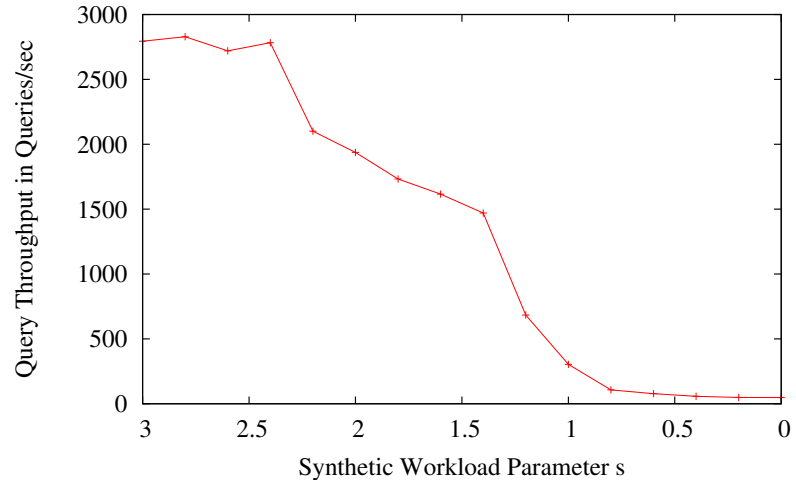


Figure 2.6: **Robustness to Variable Workload: Query Throughput**
Synthetic Read-Only, Hash Part., Vary s

"performance cliff".

We conclude that performance is reasonable even for unreasonable workloads. Adding a new, minor use case and respective queries to a Crescando installation should *not* violate latency requirements of existing use cases.

2.2.4 Recovery

We already mentioned that the Crescando Storage Engine provides a recovery mechanism which is based on logging and checkpointing. This experiment

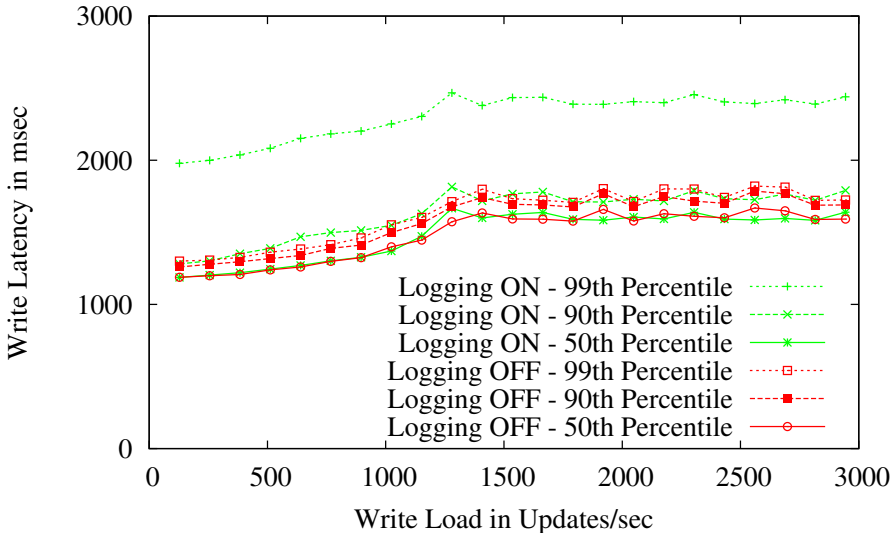


Figure 2.7: Logging Overhead under Variable Update Load; 15 GB Data

focuses on assessing the quality of the recovery scheme based on two distinct metrics. First, the overhead of logging and checkpointing during normal operation and then the time required to resume normal operation after a crash.

Logging Overhead

We measure the relative difference in write latency of the Crescando Storage Engine under an increasing write load, with logging enabled and logging disabled respectively. Write latency is defined as the time from the client enqueuing an update to the client receiving a confirmation. A confirmation is only returned after all potentially affected scan threads have executed the query, and the query has been committed to the log.

We used 15 GB of data round-robin partitioned. Since queries are not logged, and `INSERT`s and `DELETE`s would change the data volume, we use an `UPDATE`-only workload. As figure 2.7 shows, there is no impact on the 50th and 90th percentile of queries, only on the 99th percentile. This increase is not because of pending confirmations. The log is flushed multiple times a second and grows by at most a few hundred kilobytes each time. In other words, update queries are already committed to the log by the time the last scan thread finishes their execution. The measured overhead is rather the result of concurrent snapshot selects *cf.* Section 2.1.3)

Notice that the 99th percentile increases by about 700 msec with logging enabled, which is approximately the time it takes to copy 1 GB of memory from one NUMA node to another on our fully loaded test platform. Writing a snapshot of a memory segment to disk takes almost 10 seconds, whereas a scan cycle takes around 1.5 second. For our 15-way system, a full snapshot takes almost 150 seconds, during which almost 100 scans with no snapshot

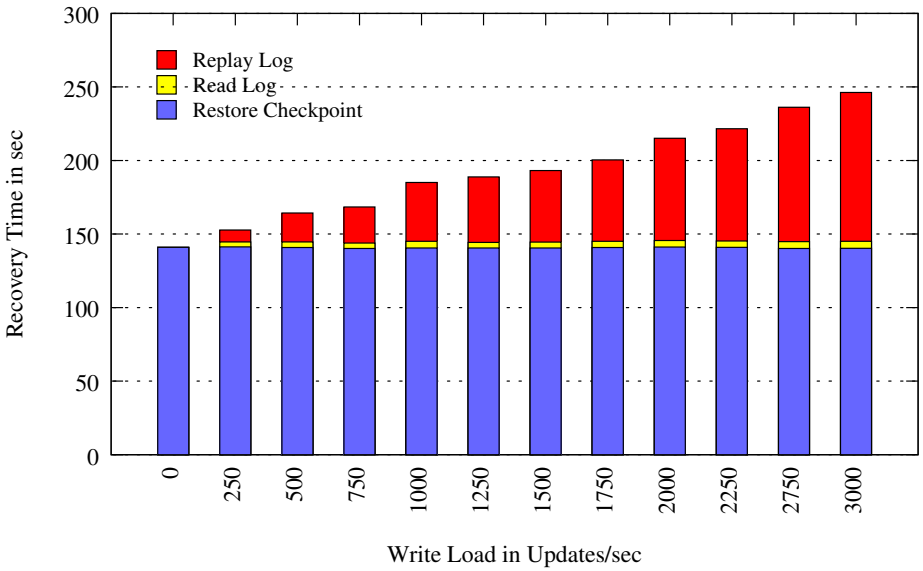


Figure 2.8: Recovery Time under Variable Write-Only Load; 15 GB Data

select active are performed. Consequently, only the 99th percentile of queries is affected in the diagram.

Recovery Time

Next, we measured the time it takes for the Crescando Storage Engine to restart and resume normal operation under an increasing write load. We let the test driver run for half an hour before abruptly terminating it with a segmentation fault signal. The data volume, partitioning, and workload composition are identical to the previous experiment.

The results are shown in figure 2.8. They match the analytical predictions of section 2.1.3. Recovery time is dominated by the time it takes to read and restore the checkpoint, which is the data volume (15 GB) over the disk bandwidth (110 MB/sec). The log itself never grows to more than 64 MB, and the time it takes to read and de-serialize its contents (the update queries) is negligible. The log replay time is proportional to the update rate and never exceeds the time to restore the checkpoint. In fact, it is lower by at least one third, because, in contrast to normal operation, memory segments always have a full input queue and need not concurrently update the statistics.

2.3 Analytical Performance Model

Evaluation of the Crescando storage engine showed that it is able to sustain any workload with *predictable* latency, varying both in terms of diversity and update rate. In order to support the predictability argument, we designed a set of experiments that allowed us to generate a set of equations that describe the Crescando storage engine's performance.

In order to take accurate results, we had to inject code into the Clock Scan algorithm, so as to control the number of concurrently executed operations and separate scan time from queuing delays. The injected code would pause the Clock Scan algorithm at the beginning of the memory segment, waiting until a specific number of operations are pending. Then it measures the time required to scan the full segment. We will mention to this time as *cycle time*.

Given the cycle time, we are able to bound query latency, provided we know the operation queue sizes. Since the pending operations queue dominates the query latency, we can estimate the worst-case query latency as $Q * the\ worst\text{-}case\ cycle\ time$, Q being the ratio of the pending operations queue size over the running operations queue size.

Variables

We begin our analysis by identifying the variables in Clock Scan's mathematical model. Most of these variables could be modified and the system's behavior was monitored as the variables changed. Unfortunately, we could not modify all variables. Variables such as CPU frequency and memory bandwidth could only be used by making certain assumptions. The memory segment size is the first variable that is going to be examined. Queue sizes, which translates into number of concurrent operations being evaluated also affects latency, as already mentioned in Section 2.1.1. Update rate is examined in the third experiment. Finally, we will examine if and by what amount parallel scan threads affect each other.

2.3.1 Memory Segment Size

In our first experiment, we measure the way memory segment size affects cycle time. For this reason, we keep every parameter of Clock Scan constant, while varying the memory segment sizes. We used a single memory segment that was enqueued with read-only Amadeus-like operations, while allowing only 512 concurrent operations. We measured the average and maximum execution times of 1,000 cycles.

The results are presented in Figure 2.9. As could be expected from a system based on scans, the memory segment size linearly affects the cycle time. A noteworthy fact is that the maximum cycle time is very close to the average cycle time, which strengthens our argument of predictability.

2.3.2 Number of concurrent operations

Next, we examine how the number of concurrent operations affects cycle time. A relatively big number allows clock scan to partition operations having *similar* predicates in indexes, which results in faster evaluation of operations. A small number of concurrent operations may not allow an index to be built, forcing evaluation of every single operation. On the other hand, a very big number of concurrent operations requires proportionally more memory, which means that operations may no longer fit in the L2 cache. In order to perform this

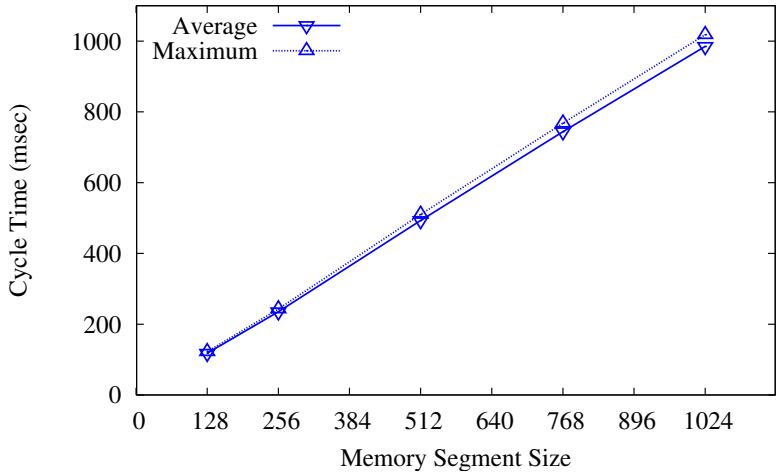


Figure 2.9: **Memory segment size against cycle time**
Amadeus Read-Only, 1 Memory segment, Vary segment size

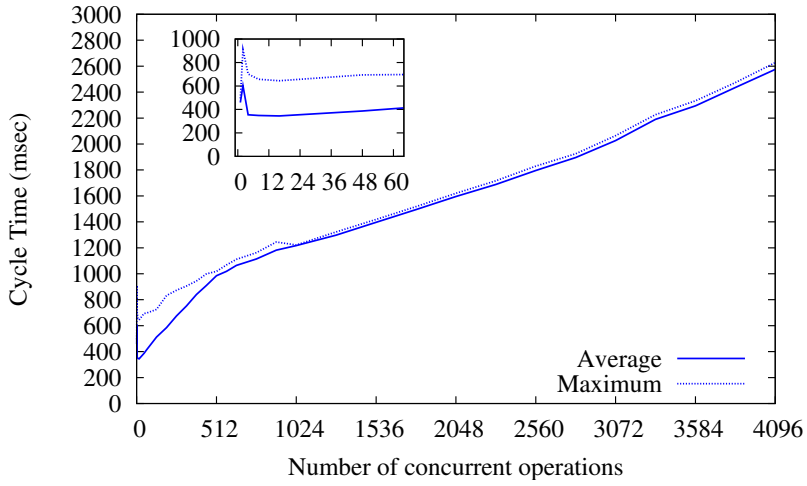


Figure 2.10: **Number of concurrent operations against cycle time**
Amadeus Read-Only, 1 Memory segment, 1 GB segment, vary number of concurrent Operations

experiment, we used a single memory segment of size 1 GB, while enqueuing read-only Amadeus operations.

Figure 2.10 shows that the number of concurrent operations affects cycle time almost logarithmically for small number of concurrent operations. This can be explained by the tree indexes that clock scan generates. The small spike, plotted in detail in the subfigure, illustrates what was described in the previous paragraph. During a memory segment scan, the number of concurrent operations can be controlled with the pending operations queue size. For a small

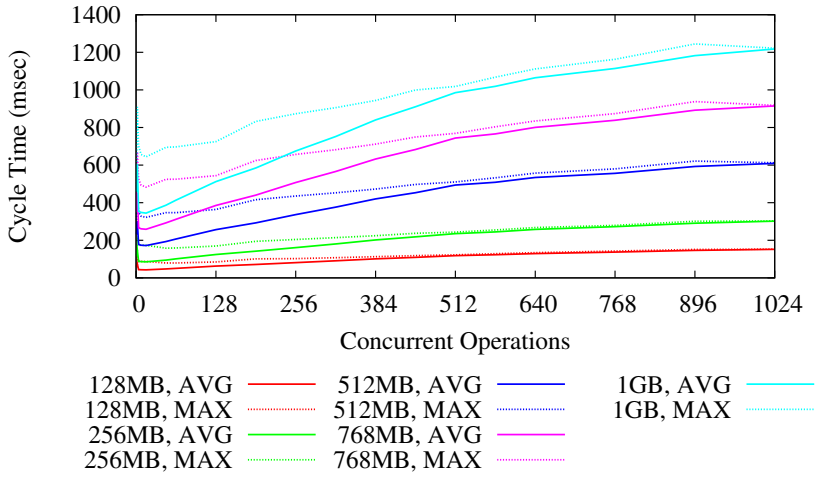


Figure 2.11: **Memory segment size and number of concurrent operations against cycle time**
Amadeus Read-Only, 1 Memory segment

number of concurrent operations (up to 1024), the cycle time is dominated by the logarithmic index probing overhead. As we push in more operations, an linearly increasing fraction of queries matches the data, resulting in a linear overhead of query evaluation.

In order to summarize how both memory segment size and concurrent operations affect cycle time we used different combinations and plotted the results in figure 2.11. Next, we tried to model these curves which resulted in equation 2.1

$$T_{cycle} = S * c_0 * (\log_2(F + c_1) * c_2 + c_3) \tag{2.1}$$

In the equation F stands for the *Share Factor* or the number of concurrent operations. S stands for the segment size, measured in MBytes, while c_0, c_1, c_2, c_3 are constants. The constant c_0 reflects the hardware capabilities of the machine. We assume that c_0 decreases proportionally with increasing sequential processing capability. c_3 reflects the time of a memory segment scan without processing queries. c_1 and c_2 reflect the cost of a query. The logarithm in the equation is only valid for a small number of concurrent operations (less than 4,096), as we know that beyond that point linear overheads of cache misses becomes dominant.

The values of the constants for our testing platform are:

$$c_0 = 0.4$$
$$c_1 = 128$$
$$c_2 = 0.66$$
$$c_3 = -4.07$$

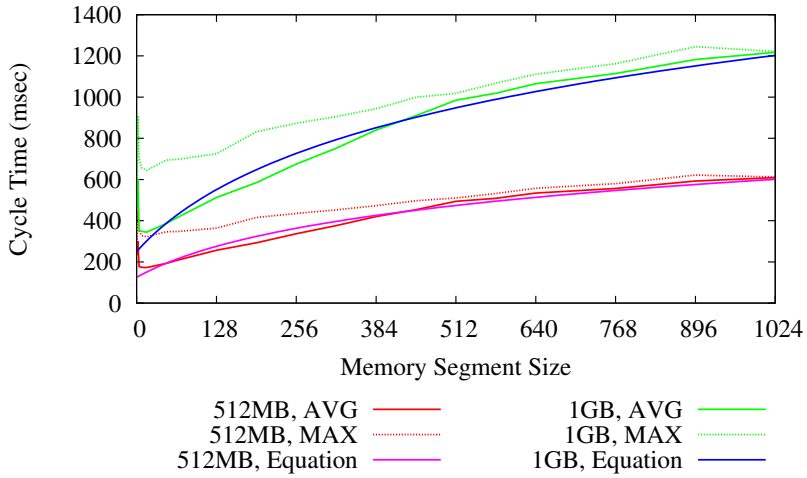


Figure 2.12: **Comparison of Analytical Model with Real Data**
Amadeus Read-Only, 1 Memory Segment

which produce the curves of figure 2.12.

2.3.3 Number of segments

In the second set of experiments, we try to measure how different scan threads interfere with each other. Scan threads were designed in a share nothing fashion. However, the operation queues, which is the only way of passing operations to a memory segments could introduce some overhead. For this experiment, we used round-robin partitioning and an increasing number of scan threads as we measure their average query throughput.

The results are shown in figure 2.13. As we can see, adding more segments causes a small, but still noticeable performance loss in query throughput. The step-like behavior is an artifact of the hardware architecture. Our hardware hosts 4 sockets, on each of which 4 CPU cores reside. The first 3 segments are allocated on processors belonging to the same socket (note, that there is an extra thread running that belongs to the benchmark driver). Once we try to allocate a fourth segment, 2 cores belonging on different sockets have to operate, resulting in this minimal performance drop.

We have modeled the given curve with the following equation

$$X_n = \frac{F}{T_{cycle}} * (\frac{1}{1 + n * c_m})^2 \tag{2.2}$$

where, X_n is the throughput of one of the n scan threads running concurrently, F stands for the share factor and c_m is the overhead inflicted per memory segment. For our case, we have calculated c_m to 0.0033. Figure 2.13 also shows the comparison of the real results compared to our model.

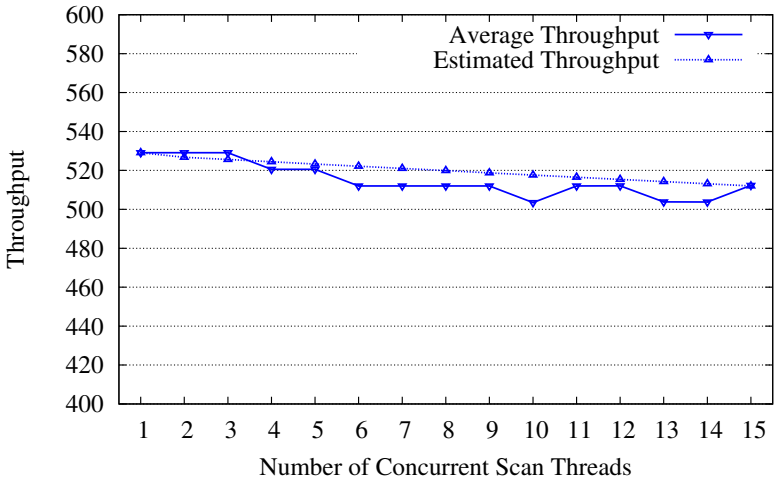


Figure 2.13: **Interference of memory segments with each other**
Amadeus Read-Only, 1 GB Cumulative Dataset, 1 GB Memory Segments, Vary Number of Segments

2.3.4 Number of segments and segmentation strategy

While these equations are straightforward, things become complicated when different segmentation strategies are used. The reason is that in hash segmentation, a query that does not match the hashing attribute has to be executed by all memory segments, increasing the *cost per query*. In round robin partitioning this effect always exists, making it easy to calculate, as every query, no matter what predicates it contains, has to be executed by all scan threads.

We have modeled the effect of the segmentation strategy with equation 2.3.

$$X_{total} = \frac{1}{w} * X_n \tag{2.3}$$

where X_{total} is the total throughput, X_n is the individual throughput of a scan thread and w is the *query weight*. The query weight depends on the segmentation strategy and can be any of the following:

$$w_{roundrobin} = 1 \tag{2.4}$$

$$w_{hash} = \frac{p + (1-p) * n}{n} \tag{2.5}$$

$$w_{k-hash} = \frac{\prod_{i=1}^{k}(p_i + (1-p_i) * n_i)}{\prod_{i=1}^{k} n_i} \tag{2.6}$$

where n is the number of memory segments and p is the probability of a query matching the hashing attribute. The w_{k-hash} is a generic form of 2.5 for multi-level partitioning, where k stands for the number of levels. In this case, n_i stands for the number of nodes in the ith level and p_i for the probability of a query matching the hashing attribute of the ith level.

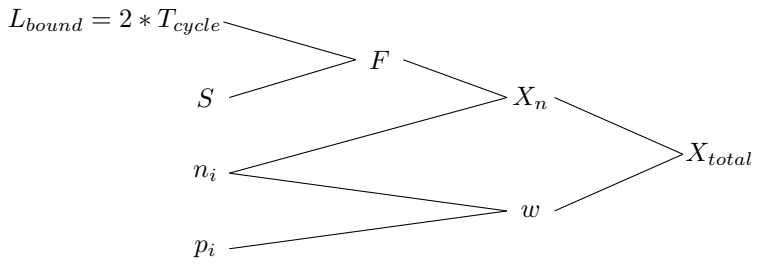


Figure 2.14: **Mathematical connection of latency to throughput**

Combining all these equations, we are able to make a connection between the expected latency of our clock scan, for a given query load and vice versa. This is illustrated in figure 2.14.

As already mentioned, there are a few parameters that are not modeled. However, we assume that a change in these parameters affects only the constant factors of our equations. Such parameters are the processor frequency and the sizes of L1 and L2 caches. For instance, running Clock Scan on a system with higher CPU frequency should result in proportionally higher throughput given a constant latency requirement.

Chapter 3

Distributed Architecture

3.1 Motivation

Chapter 2 overviewed the Crescando Storage Engine and the Clock Scan algorithm which provide a number of features: predictable latency, scale up in the number of CPU cores in a system, atomicity of operations and offline recovery. However there are certain limitations to the capabilities of the Crescando Storage Engine. In short, the hardware imposes a limit on the amount of data a node can hold and a limit on how much it is able to scale.

The Crescando Storage Engine takes advantage of modern multi-core machines and uses parallel, cooperative scans to evaluate operations. The performance of the Engine scales on the number of scans that run in parallel. However, since each scan requires one CPU core, there is a bound on the number of scans that are able to run concurrently. At the time this thesis was written, commercial of the shelf systems could support up to 32 CPU cores. We expect this number to grow in time, but in case we want to provide a solution so that our system is not bounded by any hardware limitation.

Additionally, since the Crescando Storage Engine is using main-memory to store data, the size of the dataset the engine can support is also limited by the hardware. The known limit at the time this thesis was written was 128 Gigabytes, a value almost one third of the amount required in our case study, the Amadeus S.A. booking system.

Finally, our case study requires that the system is able to provide high availability. The Crescando Storage Engine provides only offline recovery, so it cannot be used as a live system with a target six-nine availability. We would like to provide a mechanism to render the overall system almost invulnerable to single system failures by providing online fail-over.

All said, it is clear that we need more than one storage nodes, arranged in a distributed environment, where we can increase the number of CPU cores by increasing the number of storage nodes in the system. In order to be able to support datasets larger than the main memory size, the system has to partition

the dataset across nodes. Last but not least, the distributed system should be able to single node failures. Such failures can be dealt with using replication. Even though replication is not part of this thesis, the design of the system should provide the infrastructure for it, either by adding a layer on top of the system, or by altering the storage node subsystem.

3.2 Distributed System Overview

The design of our distributed system is based on three discrete layers: the storage layer, the aggregation layer and the clients layer. Each layer provides different functionality and guarantees. The distributed system should meet the requirements of our case and additionally guarantee write monotonicity. Prior to analyzing the layers that compose our distributed system, we will present the communication mechanisms that are used to connect the three layers.

3.2.1 Communication Principles

One of the key properties of our distributed system is that queries are not handled in the same way as updates. The system should guarantee write monotonicity while there is no requirement for consistent reads.

Regarding queries, the system handles them in a best-effort manner. No consistency guarantees beyond those implicit to the Clock Scan algorithm are given. Remote calls to enqueue a query are allowed to be lost or ignored if for example the storage nodes cannot cope with their current load. However, the client receives the *end-of-results* message only when *all* tuples have been sent successfully to him. A single tuple delivery failure renders the query as failed, and the client should not receive any confirmation. The client is then expected to reissue the query after some time. Additionally, the order in which tuples answering one query are received by the client is not defined and thus the client should make no assumptions on it.

On the other hand, an update operation should never be lost. Updates are propagated into the system using idempotent messages. The update issuer is expected to identify every update with a unique monotonically increasing *timestamp*. Storage nodes execute updates strictly in this defined order. Additionally, the client is expected to remember any updates that have been issued but not yet confirmed or rejected. If the client fails to receive an answer after some predefined time, he reissues the same update with the same identifier. In case a storage node receives an update that has been already evaluated, he immediately issues a confirmation message without re-evaluating the update.

All remote procedure calls between the various layers are similar to the interface of the Crescando Storage Engine: enqueue an operation and dequeue a result. However, since a distributed environment has different characteristics, the system should take into consideration cases of message loss and node failures. In terms of operations, two additional operations are supported in the distributed environment: the *abort select* and the *no-op* operations. The abort select operation, informs a node that the client is no longer interested in the

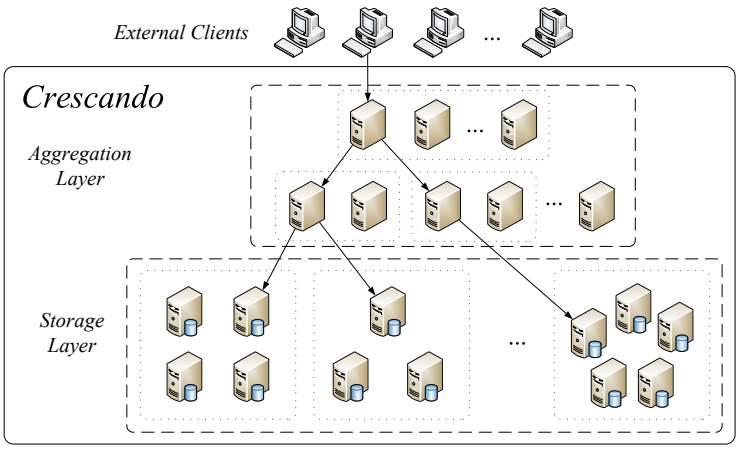


Figure 3.1: **Distributed Architecture Overview**

results of a previously issued query. Any newly discovered tuple that matches the query should not be sent to the client. The no-op operation is used in combination with an update operation. It is sent to a node that is not responsible for evaluating the real operation in order to maintain the logical numbering of executed updates.

The distributed system overview is visualized in figure 3.1. We will overview its three layers in a bottom-up fashion.

3.2.2 Storage Layer

The storage layer is, as the name suggests, the layer where the dataset resides and is comprised by a set of *storage nodes*. Each storage node runs locally the Crescando Storage Engine which was presented in Chapter 2 and exposes its interface to the network, so that it can be addressed using a RPC-like interface.

For reasons of simplicity we assume that the configuration of the set of storage nodes is static. Memory segment size, number of memory segments and share factor can only be set during start-up. The final system should be able to overcome membership issues and support dynamic node allocation.

The storage layer is also responsible for providing durability. In this master thesis' scope, we restrict our system to perform off-line recovery only in case of storage node failures. However the infrastructure for providing on-line recovery is already part of the distributed architecture design.

In order to support on-line recovery, the storage layer allows storage nodes to form groups, called *replication groups*. In a replication group, every node contains the same partition of the dataset. The *master node* of a replication

group acts as a *workload splitter* (query scheduler), which clusters and forwards incoming queries to different slave replicas based on the queries' selection predicates. This homogenizes the workload at individual replicas, increasing the effectiveness of query indexes, with a result of potentially super-linear scale-out for queries. Updates are not affected, since a replication group operates in a *read-one-write-all* fashion.

Except for load balancing and group membership decisions made by the master, replicas are completely symmetric, which is highly beneficial to fault-tolerance. In a traditional architecture, losing a view or index has a dramatic impact on at least part of the workload. In contrast to this, losing a replica in a symmetric architecture causes throughput to decrease by roughly the same, predictable degree for all queries. Analysis of on-line recovery mechanisms is an open issue that is going to be covered as future work.

3.2.3 Aggregation Layer

This middleware layer is responsible for the instrumentation and orchestration of the storage layer. It defines the way data is partitioned accross storage nodes. Our system design imposes no restrictions on how partitioning is performed. The implementation of the system decides if the mapping of tuples to storage nodes is random (round-robin partitioning) or unique (hash partitioning or partition by specific criteria, i.e. all vegeterians are stored in one specific storage node).

The aggregation layer merges ("aggregates") the results generated by the storage nodes. During merge, the aggregation layer should wait for confirmations (or rejections) from all storage nodes that were involved in evaluating an operation before sending the confirmation to the client. For updates, the confirmation is a *write confirmed* or *write rejected* message, while for queries the confirmation is a special *end-of-results* message, which declares that no more tuples are going to be sent for the given operation.

The aggregation layer may contain any number of aggregator nodes, in one or multiple levels. In a multi-level aggregation setup, an aggregator node is used to partition the dataset accross a set of aggregator nodes and merge their results. In order for recursive (multi-level) aggregation to be feasible, the interface of an aggregator layer should match the interface of a storage node.

Last but not least, our design requires that all aggregation nodes are totally stateless. The lack of state allows the system administrator to add as many aggregator nodes as required in order to support the workload. It also removes the complexity of handling aggregation node failures. The upper layer (clients layer) is expected to know a set of aggregator nodes, to which it is able to send operations. An aggregator node that fails to confirm or reject consecutive operations is considered as offline.

3.2.4 Client Layer

The client layer is the layer that generates the operations that are enqueued into the system. Clients are considered external to the system. We already

mentioned in Section 3.2.1 that clients are expected to identify every update with a unique monotonically increasing identifier. For this reason, we allow exactly one logical writer in our system. The logical writer can be comprised by multiple clients, provided that there is consensus on the total order of updates. We assume that the logical writer experiences no permanent failures and maintains a durable state. The state contains the next update identifier (timestamp) as well as every update operation that has been issued but not yet confirmed. For read-only clients (clients that issue queries only), there is no such requirement and query messages are allowed to be lost or ignored at any time.

The idea of having exactly one update client is compliant with our case study (*cf.* Chapter 1.2). The table implemented in crescando is the materialized view of an existing set of tables. In this materialized view, we have exactly one update issuer, which is the fact table maintainer; either the table storage engine itself or some propriertary middle-ware.

Chapter 4

Amadeus Workload Analysis

Amadeus S.A has provided us with a complete list of SQL prepared statements, that have been executed in their materialized views for given time window of 6 months. Those SQL queries include a variety of Oracle SQL functions, mostly date/time parsing functions and binary masks. Additionally, they have provided absolute execution numbers of each query. In order to identify the ideal partitioning attribute, we conducted an extensive analysis on the given workload.

of the most important outcomes of this analysis is the number of updates performed on the materialized view. This is depicted in figure 4.1. According to Amadeus reports, the materialized view serves 1 updates per gigabyte per second, which is verified by our data given the time window of 6 months during which the queries were captured. The extremely low volume of `SELECT` is because most `SELECT` queries on Amadeus' ticketing system query the key-store database instead of the materialized view. Since queries become increasingly more diverse, we expect that the number of `SELECT` queries will increase. This doesn't hold for any update operations.

Analysis includes investigation of the attributes used as predicates. Figures 4.2, 4.3 and 4.4 summarize the results of the analysis. Almost all queries (99.7% of them) select on `PRODUCT_ID` and `PROVIDER`. Analysis showed that these two attributes are correlated: every query asking for a specific `PRODUCT_ID` (flight number) also has a `PROVIDER` predicate and vice versa. In other words, every query asking for a flight number, also asks for the airline company.

The highly used attributes, `PROVIDER`, `PRODUCT_ID`, `DATE_IN`, `SGT_QUALIFIER` and `CANCEL_FLAG` are possible candidates for a partitioning policy that minimizes the number of affected nodes per query. All five attributes have a possibility of appearing more than 90% in all incoming `SELECT` queries. However, we need a more in-depth consideration of the data in order to design a proper partitioning scheme. For example, further analysis showed that the `CANCEL_FLAG` attribute is an attribute having only two possible values (`'T'` or `'F'`), so it makes no

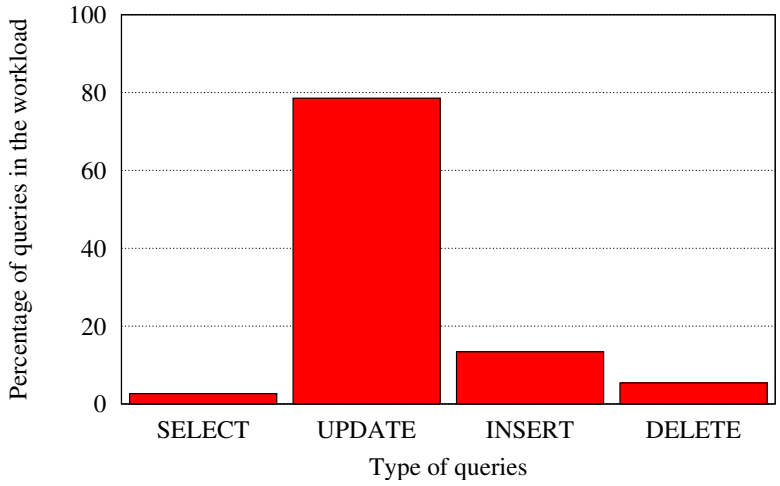


Figure 4.1: **Query Types Distribution in Amadeus Workload**

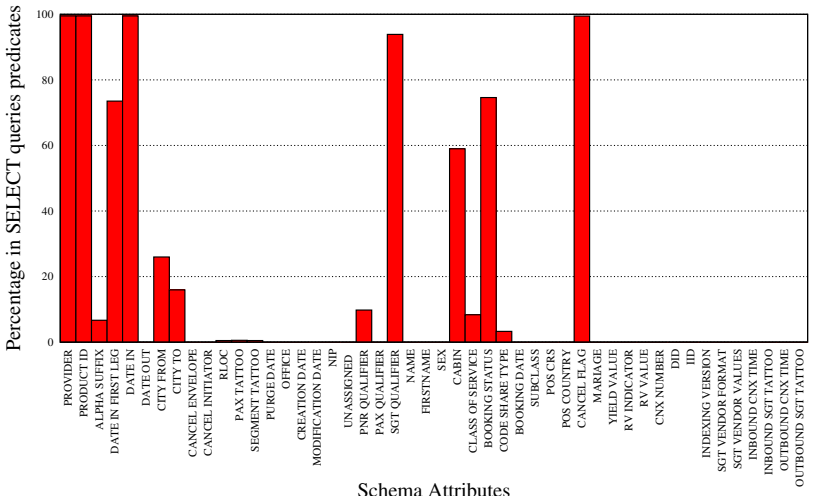


Figure 4.2: **Amadeus Workload: Predicated Attributes in `SELECT` queries**

sense to use `CANCEL_FLAG`[1] as our partitioning attribute if we have more than two partitions.

Amadeus S.A. has also provided us with a representative subset of the full dataset. It contains almost 8 million records, which allows us to extract some useful information regarding the diversity and skew of the data. Figures 4.5, 4.6 and 4.7 illustrate the distribution of different values for the most common attributes.

[1] `CANCEL_FLAG` marks cancelled bookings.

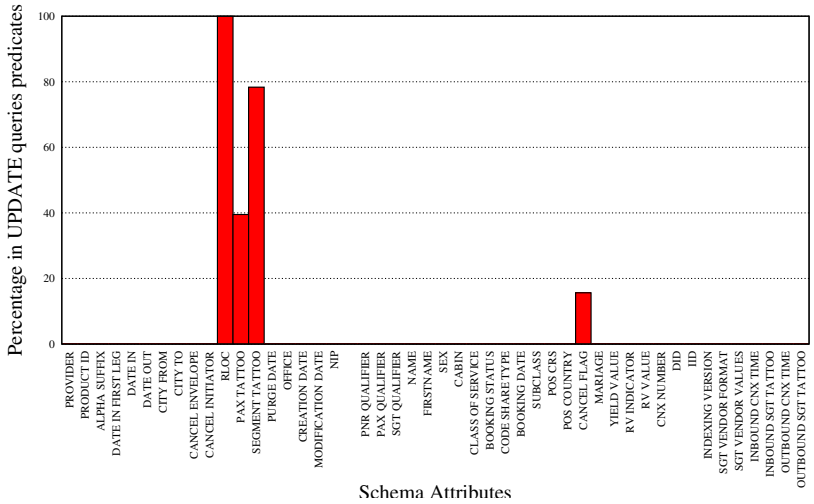


Figure 4.3: **Amadeus Workload: Predicated Attributes in `UPDATE` queries**

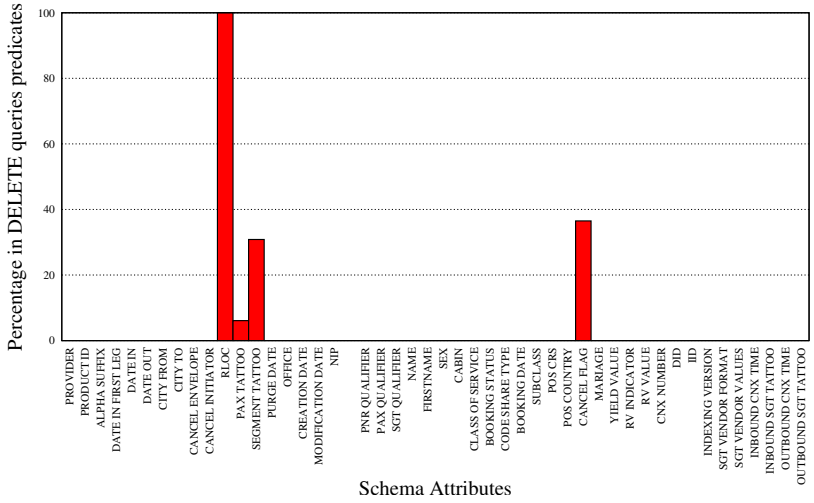


Figure 4.4: **Amadeus Workload: Predicated Attributes in `DELETE` queries**

Regarding attribute `DATE_IN`, the figure 4.6 does not capture that fact that skewness of the data is not static. In fact, we expect this skew to change every day, which makes it a problematic candidate for partitioning. To make matters worse, most queries predicating on `DATE_IN` ask for a range of values, making it a hash partitioning futile. Range partitioning on the other hand is complicated by the fact that the data skew moves over time.

For the update workload, we can make a very useful observation by examining the high usage of the attribute `RLOC`. This string attribute is used as the primary key in the PNR storage of Amadeus; every distinct `RLOC` is mapped to a small

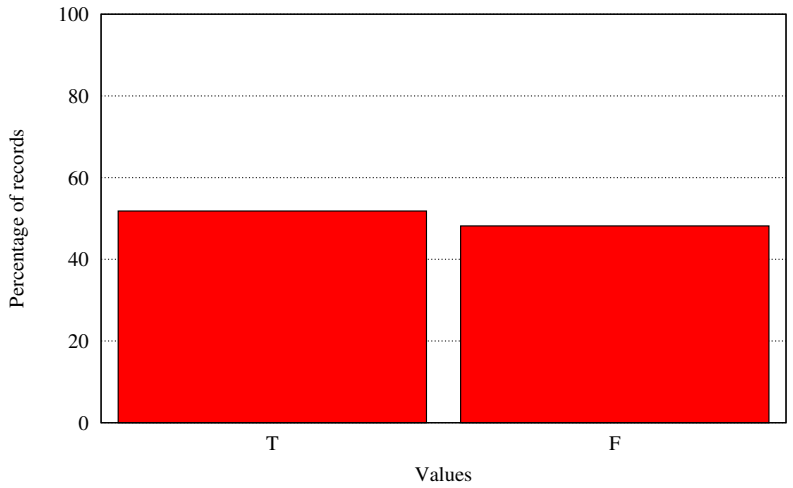


Figure 4.5: **Value distribution in attribute `CANCEL_FLAG`**

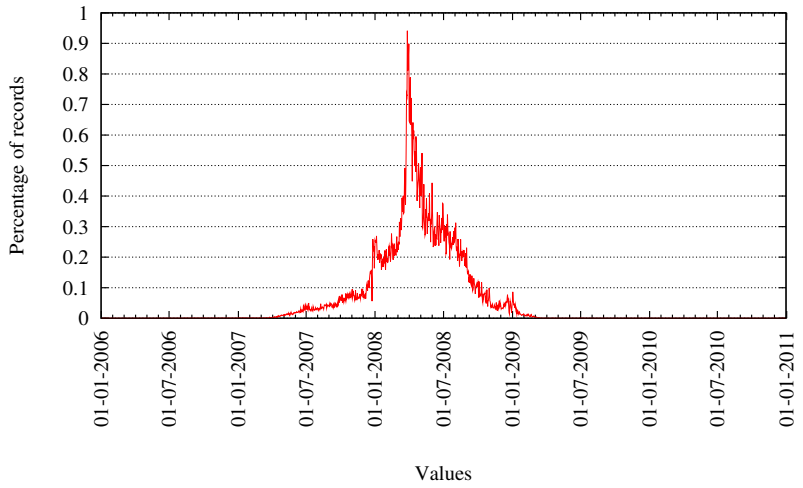


Figure 4.6: **Value distribution in attribute `DATE_IN`**

set of tickets, i.e. tuples in the Ticket view. Choosing `RLOC` as our partitioning attribute, allows us to balance the updates across storage nodes, which is crucial under a high update load.

To put some numbers on the previous case, considering the fact that the system receives 1 updates per gigabyte per second and given that the dataset is 300 gigabytes, we should expect 300 updates per second at the aggregation layer. If `RLOC` is not part of the partitioning policy, that would mean that every node has to execute 900 updates every second. Of course, 300 updates per second is significantly lower than the measured limit of clock scan (2,300 updates per second), however, we should consider some other parameters. First of all, the dataset is subject to grow and as it grows, more updates would be enqueue in

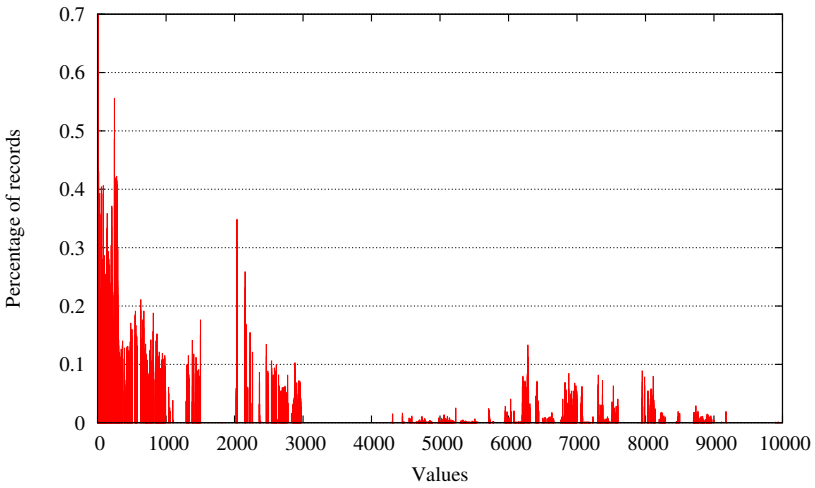


Figure 4.7: **Value distribution in attribute `PRODUCT_ID`**

the system. Secondly, the update rate is subject to grow, as more and more clients are using the system. And finally, the 1 update per gigabyte per second is an average load. We expect bursts of 6 times more updates than the average, that would go very close to the system's limit. Thus partitioning by `RLOC` is a necessity.

To summarize, the attributes that can be used in out partitioning policy are very limited, compared to the 47 total attributes. `RLOC` is consider an important attribute and should be used in order to support the update workload of the system. Additionally, either `PRODUCT_ID` or `PROVIDER` can be used as a second partitioning attribute, since these appear in almost all `SELECT` queries.

4.1 Partitioning strategy

The analysis of the workload provided by Amadeus revealed three important attributes in the schema: `RLOC`, `PRODUCT_ID` and `PROVIDER` (unique e-ticket id, flight number and airline company). These three attributes offer great characteristics for partitioning the dataset. There are different scenarios and setups in which these three attributes can be arranged in order to maximize the performance of the distributed system.

As mentioned in Section 2.1.2, the storage nodes offer an extra layer of partitioning the dataset into memory segments. Given that we have three attributes that can be used in partitioning the dataset and that one of these has to be used by the storage nodes, we are able to use two attributes in our partition strategy. This can be done using a two level hash partitioning on the two attributes. Choosing which attribute is used in which level can be determined if we know the absolute number of reads and writes that we expect in our system. If updates dominate the workload, then it is more beneficial to use `RLOC` as the upper level partitioning attribute and `PRODUCT_ID` or `PROVIDER` as the lower level.

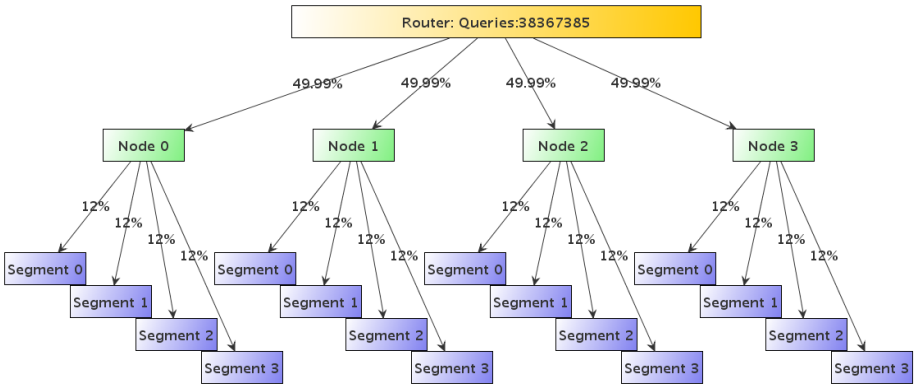


Figure 4.8: **Overview of the distributed systems and load distribution for `SELECT`s**

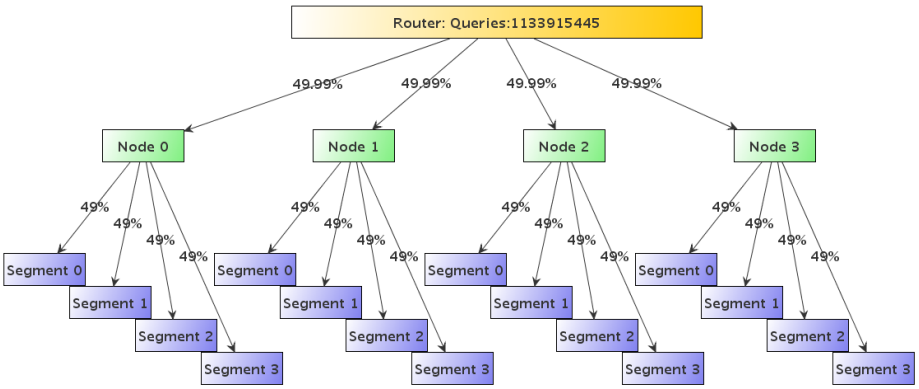


Figure 4.9: **Overview of the distributed systems and load distribution for `UPDATE`s**

A distributed query execution simulator has been implemented and the suggested partitioning scheme has been used to create figures 4.8 and 4.9. The number of nodes and number of segments in this case is set to only 4 which is already quite low. We expect the number of segments per node to be at least 15 and the number of nodes around at least 20 in order to support the 300 GB dataset.

Chapter 5

Implementation

Daedalus, as described in Chapter 3 has been implemented as a set of shared libraries for 64bit POSIX systems written is gcc-style C++. A remote client has also been implemented in Java. The remote client is able to reproduce the workload of Amadeus. This chapter will present the libraries that comprise Daedalus.

The overview of the implementation is presented in figure 5.1. The white blocks represent the pre-existing implementation of Clock Scan, the light blue blocks the libraries that implement the distributed system, while the light red blocks represent java libraries that comprise the remote client.

5.1 Communication

We will start our overview of the implemented system with the networking library. This library defines the message layer which is used by all nodes in Daedalus to exchange application-level messages. The message layer models an asynchronous network. In other words, messages may be delayed arbitrarily, lost (= delayed indefinitely), and delivered in any order. However, it does make certain abstractions and guarantees:

- Messages are delivered either complete and untampered, or not at all
- Messages are delivered *at-most-once*
- Messages may be of arbitrary size and content

These abstractions and guarantees are necessary and sufficient to implement asynchronous protocols such as Paxos at the application level. The message layer is built on top of UDP. The first and last point are improvements over "raw" UDP datagrams. Providing them is the main purpose of this library. Messages are opaque as far as the message layer is concerned. In other words, if an application sends the same string of bits twice, this will be interpreted and transmitted as two distinct messages.

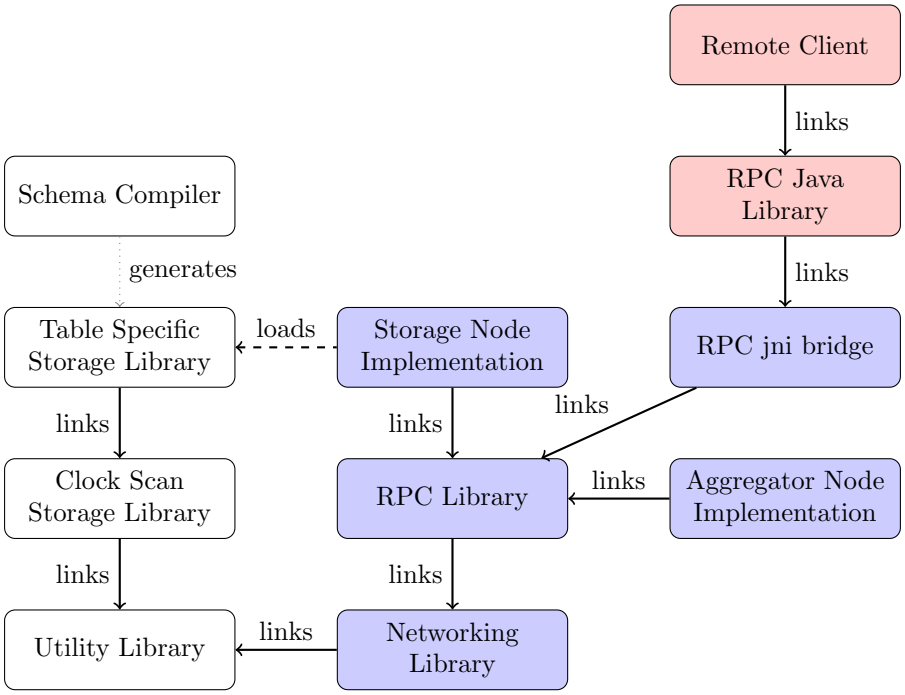


Figure 5.1: **Overview of the implemented libraries**
Light blue libraries are implemented in C++, while light red are part of the client implementation and were implemented in Java

The networking library appends a header before the message (payload) prior to sending it. Cyclic redundancy check based on CRC-64-ECMA-182 [3] ensures the integrity of both the header and the payload. In case the message is bigger than a UDP datagram, the library splits the message into fragments at the sender side and joins these fragments at the receiver side. Messages are given a unique identifier and fragments are numbered in order to ensure that the correct fragments are used to compose a message, since fragments of different messages may be interleaved.

The specific library implementation rejects reordered fragments to avoid otherwise necessary buffering. If a fragment with a different identifier than the expected one is received, the whole message is considered corrupted and is discarded. Our decision to not allow fragment reordering is based on the fact that the system is expected to be deployed in a local area network, where packet reordering is very rare.

Messages and fragments can be of variable size, however the fragment size is limited by the UDP protocol limitation where the maximum datagram size is 64 KBytes. We further limit fragment size to 1,400 bytes so that fragments can be sent as a single ethernet frame (the MTU of ethernet is traditionally 1,500 bytes). The library provides the functionality to send multiple messages at once, which reduces the number of datagram packets required to transmit

them. In this case, two or more messages may share the same datagram.

The networking library also provides the functionality of sending messages over TCP connections for purposes of flow control and reliable delivery only. In other words, there is no concept of a "session" attributed to a TCP connection. TCP connections may be broken without affecting the state of the system.

The library handles TCP connections as one-way communication channels (source-sink pairs), where a sink is not able to send a message to a source. This simplifies the implementation, but also does not allow the sink to distinguish between a source crash and some delay in response. In case of source failure, the sink may eternally wait for a message from the client. In order to overcome this issue, a separate thread is used to manage the active TCP connections and ensure they are still alive. On the sink side, this thread will send keep-alive messages to every source in predefined intervals. On the source side, the thread will receive any incoming keep-alive messages.

Since no real message can be sent from a TCP sink to a TCP source, any received message can safely be discarded, as it can only be a keep-alive message. Note that there is no need for the source to reply to this keep-alive message, as TCP does this implicitly. If the connection is broken, the sink will receive an error (SIGPIPE, broken pipe). As for the source, there is no need to send keep-alives to a sink. In case of a sink failure, the source will discover this the next time it attempts to send a message.

Keep-alive messages are messages of size 1 byte, which including the MAC, IP and TCP headers sums up to 59 bytes. Given that, we consider the network overhead of sending keep-alives minimal and in fact we are able to send thousands keep-alive messages per second. In order to use a proper keep-alive interval, we had to consider the latency requirements of our case study. By default, Daedalus sends keep-alives every 500 msec which means an unresponsive node can be discovered in less than one second.

5.2 Serialization

In order to be able to transmit operations and results between nodes, we defined a serialization scheme that is implemented inside the RPC library. The RPC library is built on top of the Networking Library, as shown in figure 5.1. The library serializes the operation or result to an array of bytes (= message) and then uses the networking library to send them. On the receiving side, for every received message, it tries to deserialize it and then passes the deserialized Crescando operation or result to the upper layer.

Concerning the choice of protocols, UDP is sufficient for sending operations over the network and receiving update results. For query results, we resort to TCP, since we expect that a query may generate a vast number of results in unpredictable bursts. For this purpose, we rely on TCP's built-in flow control and acknowledgment schemes.

The library is able to serialize all different types of crescando operations: `SELECT`, aggregate `SELECT`, `UPDATE`, `DELETE`, `INSERT` and abort `SELECT`. The first five operations are regular unnested-SQL queries, while the abort `SELECT` operation is a hint to the storage nodes to stop *streaming* result tuples to the client. Operations are grouped into queries and updates, as they have to be treated differently. In crescando, queries are treated in a best-effort manner, meaning that a query may be lost or ignored without affecting the system state. However, a lost update breaks write monotonicity. For this reason, updates contain a special *timestamp* field, which defines a total order in which updates should be evaluated and allows the receiver to detect lost updates. The library introduces a special operation, the NO-OP, which is used in combination with an update. In order to guarantee consistency, all storage nodes have to be in the same timestamp. An update directed to a specific storage node must be followed by NO-OP operations to all other nodes, in order to maintain the logical numbering of executed updates. This allows a storage node to identify a missing update from an update that was not intended to be executed on this node.

For operation results, the library also makes a distinction between query results and update results. We already mentioned that query use TCP to send results, in order to take advantage of the built-in flow control. Additionally, the use of TCP connections guarantees that no individual result tuples are lost, something that would render the query results as incomplete. Every query result contains an identifier of the operation and a projection of a single tuple that matches the query predicates. When an operation is complete, a special *end-of-stream* result is send to inform the clients. In order to improve performance, the library allows sending multiple query results in the same TCP packet.

On the other hand, update results are sent via UDP datagrams. Packet loss is not an issue for correctness, as the client is able to re-send the update without hurting consistency (*cf.* message idempotency). An update result contains the logical timestamp of the update that was executed. In case of an error, the result also contains the description of the error. Such errors could be that the operation has a non-expected timestamp, indicating packet-loss or that the update could not be enqueued into the storage nodes because of high load.

The RPC library extends the networking library, externalizing it's channel maintenance functionality to the upper layer and augmenting it with message serialization/deserialization. Extensive presentation of the message layout can be found in appendix A.

5.3 Crescando Storage Nodes

Crescando storage nodes have been implemented as C++ executables, built on top of the networking and the RPC libraries. The storage node implementation is expected to run on a multi-core system, where all but one core run the Clock Scan algorithm. The last core is free to poll operations from the network and send results to the clients.

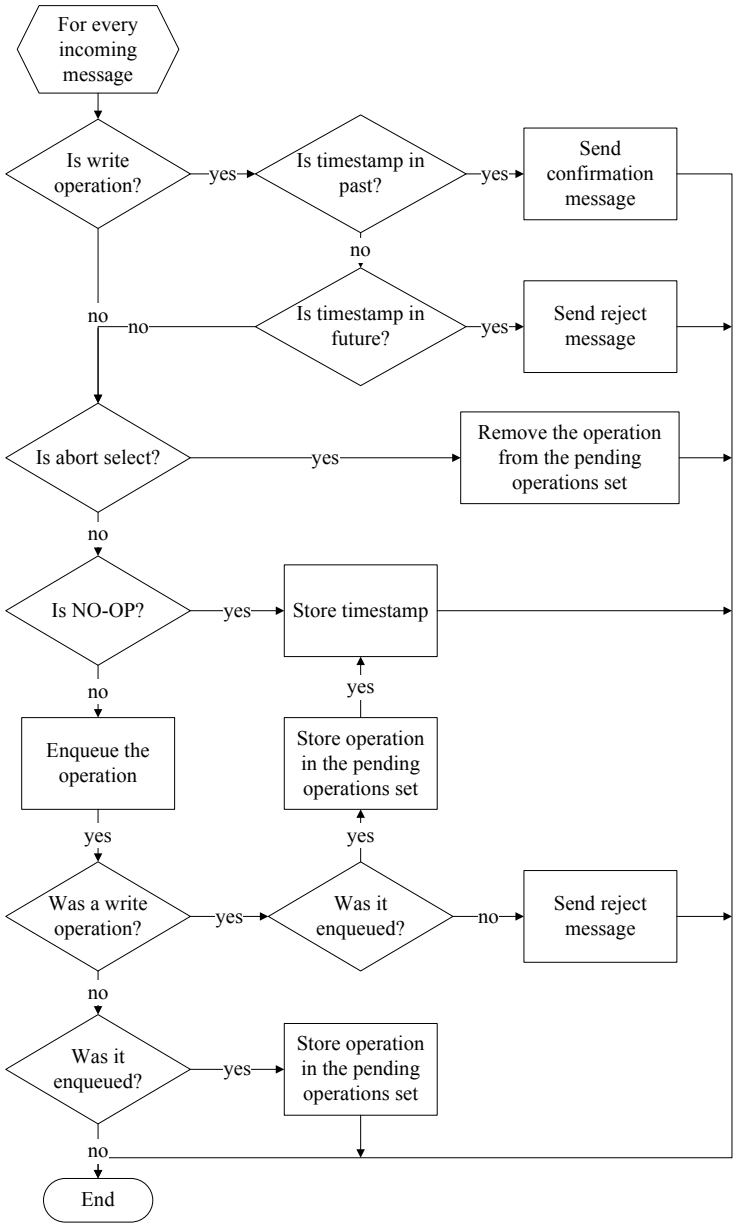


Figure 5.2: **Storage node algorithm flow chart for handling incoming messages**

In figure 5.2 the flow chart of the algorithm that handles incoming messages is presented. It is clear that the path for handling updates is longer compared to the one for queries. The implementation maintains a set of "pending operations", which are operations that are currently being evaluated in the Crescando

Storage Engine. Additionally, it keeps track of the timestamp of the last executed update. For updates, it distinguishes three cases:

- **Incoming update has a timestamp in the past:** In this case the storage node has already executed the incoming operation, so a confirmation message for the specific operation is issued (messages are idempotent).
- **Incoming update has a timestamp in the future:** This occurs only when some updates failed to be delivered to this node. The storage node issues a reject reply message[1] which includes the timestamp that was supposed to be received.
- **Incoming update has the same timestamp as the storage node:** The storage node tries to enqueue the operation into the controller. However, the storage node's timestamp is increased only if the operation was successfully enqueued. The controller may refuse to execute the given operation, in case it is not able to handle the load. If the controller refused to enqueue the operation, the storage node issues a reject reply message as a hint to the client to decrease the load.

No confirmation message is issued in case the update is successfully enqueued into the controller[2]. The algorithm stores the update in the pending operations set and the confirmation is only sent when the controller returns the result of the specific operation. Increasing the timestamp prior to getting the result does not hurt write monotonicity since Clock Scan executes updates strictly in the given order. So, while an update is being evaluated, other updates may be received and as long as their timestamp is monotonically increasing, they will be also enqueued into the controller.

Handling a query is much simpler. Upon receiving a query, the storage node tries to enqueue it and in case it succeeds, it adds it to the pending operations set. No replies of failures while enqueueing are issued. Abort `SELECT` is handled differently: the storage node removes the operation identified in the abort select from the pending operations set. This is sufficient to stop the storage node from streaming query results to clients, since the pending operations set is checked prior to sending a query result.

While receiving and handling remote operations, a storage node tries to dequeue any result from the Crescando Storage Engine and send it to the client node that issued the operation related to the query result. The flow chart of the algorithm is illustrated in figure 5.3.

Once a result is dequeued by the storage node, the pending operation set is probed and the operation related to the result is retrieved. This is achieved by means of a unique identifier that every operation carries and is also part of the result. If the operation does not exist in the pending operations queue, the result is discarded.

[1] Reject messages are an optimization and not necessary for correctness
[2] Though that would be another possible optimization

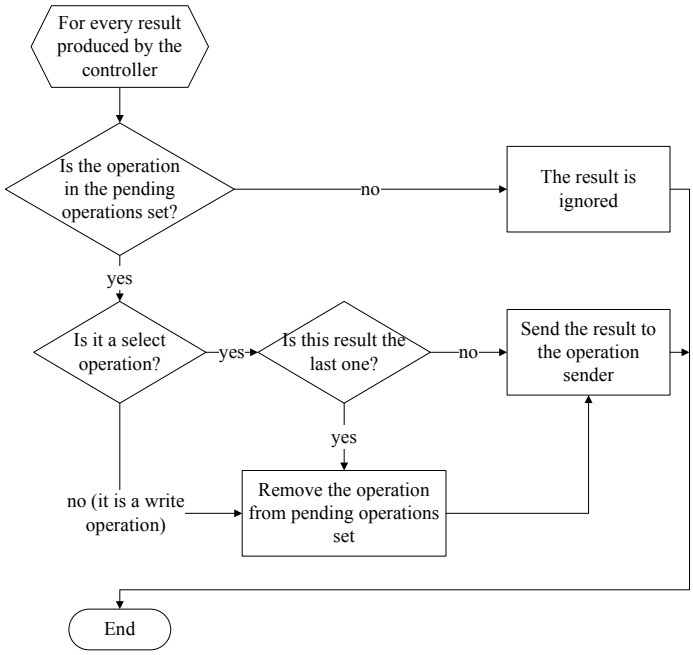


Figure 5.3: **Storage node algorithm flow chart for handling results**

As with incoming operations, query results are treated differently compared to update results because queries may return more than one results, while all updates return exactly one result. There is also a difference in the transport protocol, as TCP is used for query results and UDP for update results. For query results, the algorithm checks if the result is the end-of-stream result and if this is the case, it removes the operation from the pending operations set. In any case the result is serialized and sent to the source of the operation using the RPC library. Note that the source of the operation is defined in the RPC operation semantics.

5.4 Aggregation Layer

As already discussed in Section 3.2.3, the aggregator nodes are responsible for partitioning the dataset across storage nodes and routing operations accordingly. The way tuples are assigned to storage nodes should minimize the effects of skew in the data, i.e. a heavily skewed dataset should create an as much as possible symmetric load on the storage nodes.

Additionally, the dataset should be partitioned in such a way that we don't have to forward every operation to every partition of the dataset. This means we do not only partition data across nodes, but also operations. Load balancing becomes an issue, since we want to avoid having all queries being answered by one specific node.

Designing a partitioning policy that deals with both dimensions of the problem (data and workload partitioning) requires analysis of the workload that the system has to deal with. The analysis we conducted in Chapter 4 helps us decide on a proper partitioning strategy.

5.4.1 Implementation Details

The aggregator node has been implemented as a C++ executable. The query routing algorithm runs on a single thread, however the design of the system allows deployment of many aggregator nodes on a single machine. The implemented executable uses the networking and the RPC libraries to exchange messages with the storage nodes and the clients.

The aggregator node performs busy loops until any message is received. To simplify the implementation, we used two different listening ports, one for messages originating from storage nodes (or in general any *lower* level node) and one for messages originating from the clients (or in general any *upper* level node). By design, we know that the lower level nodes can only send query results, write confirmations or write reject messages, while the upper level nodes can only send operations.

The implemented executable keeps track of any operation that has been forwarded to the lower levels but for which no reply has been received. This "pending operations" set includes a timeout variable in order to clean-up any operations whose results failed to be received. A pluggable "partitioning policy" is used to calculate the storage nodes that should be involved in evaluating an operations. The algorithm is split in two parts.

Client Messages The first part handles messages received from clients and is presented in 5.4. First it examines if the incoming operation is an abort `SELECT` operation. If this is the case, it forwards the abort request to the proper storage nodes and removes the operation from the pending operations queue. If it is not an abort `SELECT` operation, the client sends the operation to the nodes that are responsible for answering the operation and adds the operation to the pending operations set. Finally, the algorithm checks if the operation was an update and if so, it sends a NO-OP operation to every storage node that is not supposed to answer the operation.

The implementation provides three different pluggable partitioning policies: round robin, hash and dual-hash. The round robin policy places incoming records (`INSERT` operations) in storage nodes in a round robin fashion. For all other operations, all the storage nodes have to be involved in evaluating them. The hash policy places incoming records on storage nodes based on the hash value of some attribute of the record. Any incoming operation having an equality predicate on this attribute will be answered by a single node. However if such a predicate is not given, the operation will be evaluated by all storage nodes.

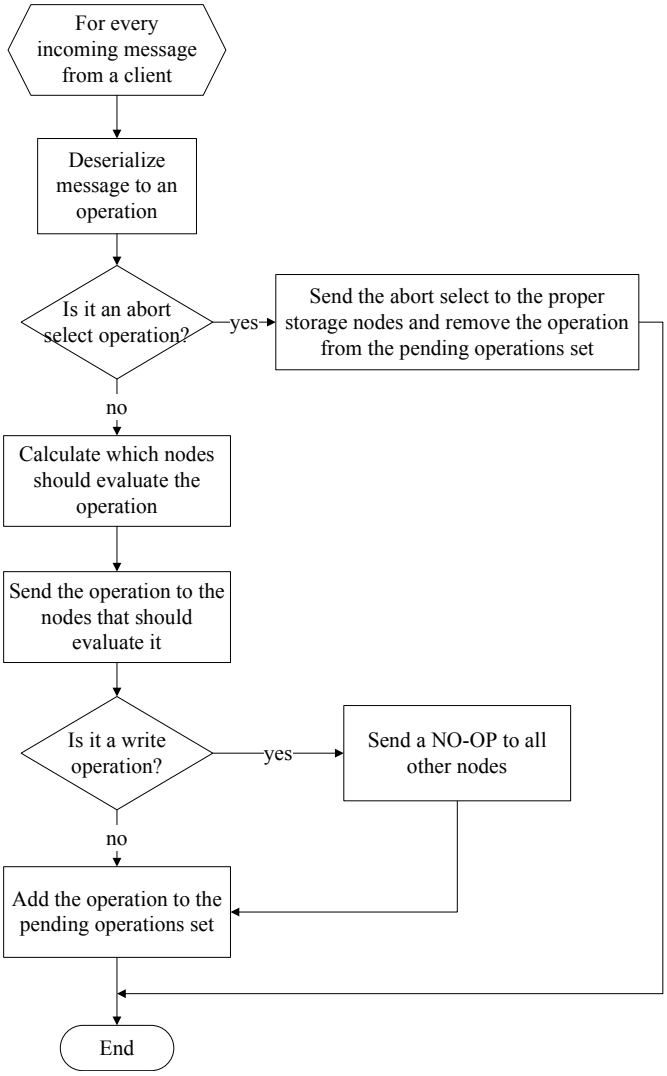


Figure 5.4: **Aggregator node algorithm flow chart for handling client messages**

We have extended the idea of the hash policy into a dual-hash policy, which uses the hash values of two attributes. Storage nodes form a two dimensional grid where the x dimension is determined by the hash value of the first attribute and the y dimension by the hash value of the second attribute. Operations that have an equality predicate on both attributes will be evaluated by one storage node, while operations having an equality predicate on just one attribute will be evaluated by either a horizontal or a vertical slice of the grid. In the case where no predicates on either attribute exist, the whole grid will be involved in answering the operation.

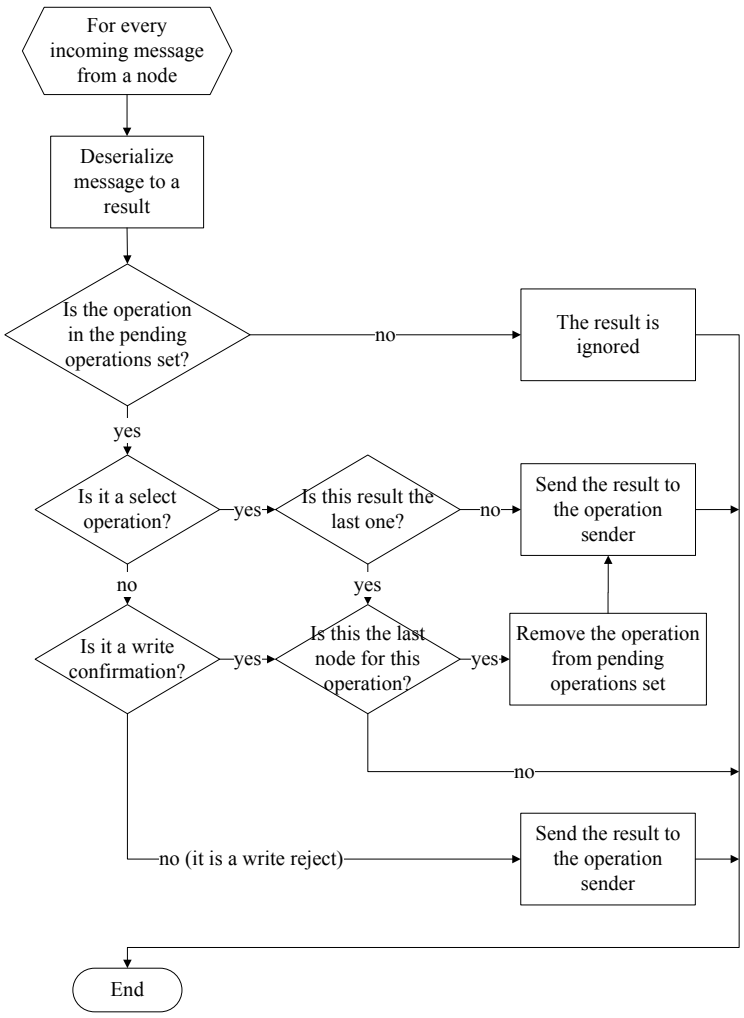


Figure 5.5: **Aggregator node algorithm flow chart for handling storage node messages**

Dual hash can be extended into a generic k-hash policy. However, as k increases we introduce more dimensions in partitioning which requires exponentially increasing number of storage nodes. Thus, we restrained to use only up to two dimensions.

Storage Node Messages The second part of the aggregator logic is in charge of receiving incoming messages from storage nodes (or in general any *lower* layer) and forwarding them to the clients. Figure 5.5 illustrates the flow chart of this process. At the beginning the aggregator node deserializes the message to a result. Since the message originates from a storage node, we know by design that it is going to be a result (a query result, a write confirmation or a

write rejection). In crescando, every result includes the unique identifier of the operation that produced it. Using this identifier, the aggregator node probes the pending operations queue. In case no operation with the given identifier is found, the result is discarded.

There are various reasons why the aggregator may not have an operation stored in the pending operations queue. The most obvious one is that an abort `SELECT` was executed, however the storage node either received the abort with some delay, or did not receive it at all (abort `SELECT`s are communicated using UDP). Other cases may be that the aggregator crashed and immediately recovered. By design, the aggregator nodes have no persistent state which means that there is no way for an aggregator to retain the pending operations set.

The algorithm then checks the type of the incoming result. In case it is a write reject, it immediately sends it to the client. On the other hand, write confirmations are not sent to the client, except if it is the confirmation from the last storage node responsible for answering the operation. Select results are handled similarly: all results containing tuples are immediately sent to the client, however the special *end-of-stream* result is only sent if it is the end-of-stream of the last storage node responsible for this query.

5.5 Remote Clients

As already discussed, our distributed system design requires that there is only one logical write client in our system, which fits to our case study, which is an implementation of a materialized view. In a materialized view, updates are issued only be the tables that constitute the materialized view. Having a single write client is key to write monotonicity, as the client is able to define the order in which writes have to be executed.

Our implementation concludes with a small client-side library built in Java, that allows clients to send operations to the system and receive results. The library keeps track of the number of executed updates and defines the order of them. Additionally it keeps track of all operations that have not yet been replied in a "pending operations" set.

The flow chart of the process for handling incoming results is illustrated in figure 5.6. The algorithm first deserializes the incoming messages to a result and then checks if the operation related with this result is in the pending operations set. If it is not, the result is discarded. If it is pending, then an action is performed based on the type of the result.

If the result is a query result, the result is passed to the application. In addition, the last query result, the end-of-stream result removes the operation from the pending operations set. For write confirmation results, the behavior of the implementation is the same: it removes the operation from the pending operations set and passes the confirmation to the application. For write reject results, the library retrieves the operation from the pending operations queue and resends it.

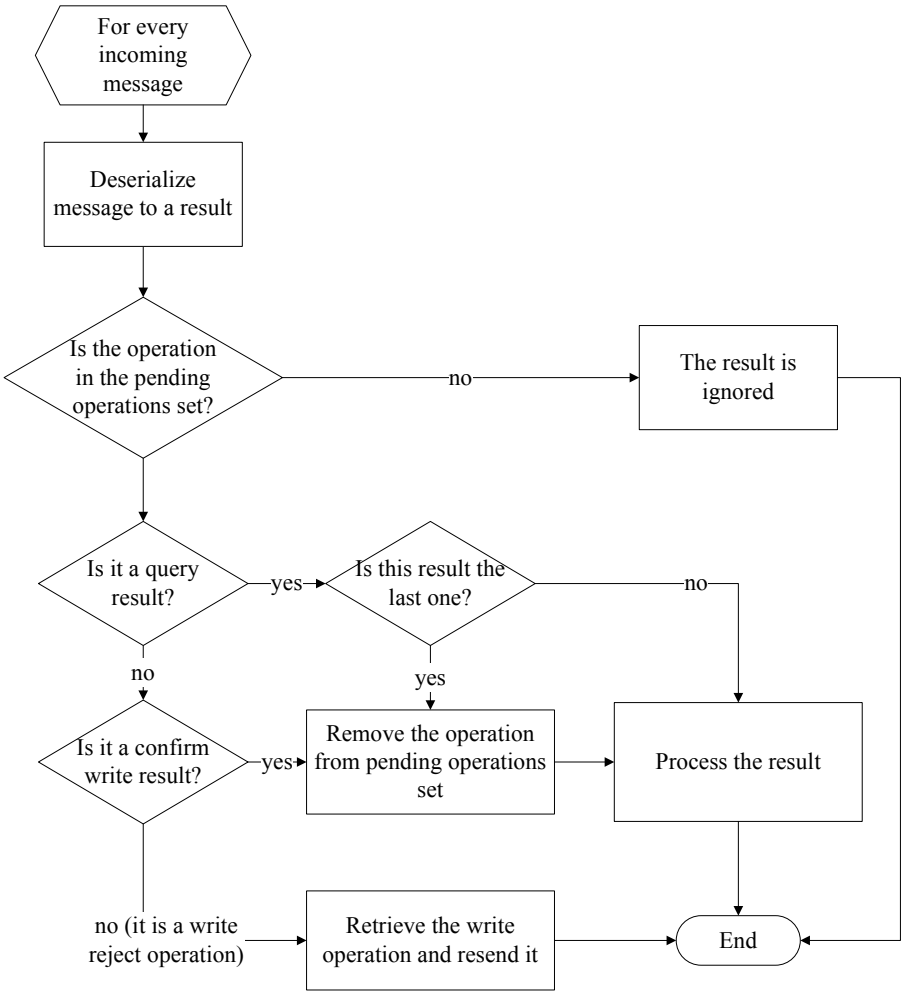


Figure 5.6: **Client node algorithm flow chart for handling incoming messages**

Chapter 6

Evaluation

6.1 Metrics

Prior to evaluating the system, we will define what we consider a valid experimental result and what not. In Section 1.2.1 we presented the requirements of our use case. Concerning data freshness, the system should apply every update in 2 seconds, thus giving us an upper bound for the update latencies. Additionally, we know that we expect on average 1 update per gigabyte per second.

Here we summarize what we measure in our experiments and what they represent:

- **System throughput** is the number of operations the system executes per unit of time.
- **System latency** measures the time required for an operation to be completed. We are interested in the median, 90th and 99th percentile latencies.
- **Local throughput** is the number of operations executed locally on a specific node of the system.

6.2 Variables

In Chapter 2, we analyzed the different parameters of the Crescando Storage Engine and how these affect the performance of a single storage node, both in terms of throughput and latency. The distributed environment has different requirements. In this section, we are going to analyze the configurable parameters of Daedalus:

- **Number of storage nodes:** Defines the number of storage nodes that comprise the system. For purposes of simplicity, we assume that all storage nodes use the same configuration (number of memory segments, segments

size and queue sizes). Each storage node stores a different partition of the dataset as in this first prototype we provide no replication in our system.

- **Storage node capacity:** Defines the amount of data that one storage node can handle when at full capacity and is equal to the the size of memory segments times the number of memory segments. We have already analyzed the effect of the memory segment size in Section 2.3, so for reasons of simplicity, we will keep this number constant in the experiments that follow.
- **Clock Scan configuration:** Defines the optimizer algorithm that clock scan uses and the incoming operation and outgoing result queue sizes. Since our mathematical model, presented in Section 2.3, allows us to predict the behavior of a node given it's configuration, we will not vary these parameters. As a reference, we used the index union join optimizer and allowed up to 256 concurrent query executions.
- **Number of aggregators:** The number of aggregators that are present in the system affect the number of queries that can be routed from the clients to the storage nodes. As the flow of operations and results increases, the aggregator layer may become the bottleneck of the system. Adding more aggregators may help in getting bigger query throughput in such cases.
- **Query rate, update rate:** These two variables define the rate at which we inject operations into the system.
- **Selectivity of queries:** The selectivity of a query defines the fraction of the dataset that matches the given query. These tuples have to be serialized into query results and propagated from the various storage nodes to the client nodes.

6.3 Testing platform

We used a cluster of 14 machines to perform our experiments. Each machine is equipped with two quad-core K10 AMD processors and 16 GBytes of main memory. Each core has 2.3 GHz clock frequency, 64 KB + 64 KB (data + instruction) L1 cache and 512 KB L2 cache. The machines are equipped with fully switched Gigabit Ethernet and are running Red Hat Enterprise Linux Server release 5.3 (Tinkanga).

In this setup we are able to use only 6 cores as memory segments. The 7th core is dedicated to the Controller thread which enqueues operations to the crescando storage engine. The last core is dedicated to receiving incoming network traffic and sending keep-alive messages periodically in order to ensure that all TCP sockets are connected. Future versions of the system will bind these two maintenance threads into the same core.

We use 12 machines as our storage nodes, one machine as an aggregator node and one machine as a client node. Note that since the aggregator nodes and client nodes use only two threads, one for the networking and one for the logic, we are able to deploy up to 4 instances of them per machine.

6.4 Experimental Results

6.4.1 Scalability

The first set of experiments focuses on how Daedalus scales both vertically and horizontally. Vertical scaling (scale up) involves adding more resources to a single node of the system. In our case, this is achieved by adding more memory segments and thus using more CPU cores. On the other hand, horizontal scaling (scale out) involves adding more nodes to the system.

In our experiments, we use a constant segment size. This means that we increase our processing capacity by adding more cores while also increasing the size of the dataset. Linear scaling is achieved when the performance of the system retains the same as we add more and more cores.

The most important difference compared to the evaluation of the Crescando Storage Engine in Section 2.2.3 is that we are not able to saturate the system with operations. In the single node system, the client receives some feedback if the operation failed to enqueue while this doesn't happen in the distributed system. The client can flood the network with queries that will be dropped as they cannot be handled and will interfere with query results. Note that queries are sent via UDP which provides no flow control or reliable delivery.

Additionally, in a distributed system setup there is no control over which operations are propagated to which storage node. In a local system, the Controller will only enqueue an operation if there is enough size in the queues of all memory segments that should be involved in answering it, as it is able to check the sizes of the incoming operations queues. In the distributed setting, such information is not provided to the aggregation layer. The aggregator nodes blindly forward each operation to the storage nodes that have to answer them. Since storage nodes are not synchronized and load balancing is not perfect, an operation that is send to two different storage nodes may only be enqueued in one of them. By design, this operation is considered as failed.

The effect described is becoming more pronounced as one hits the limits of the system. In the worst case scenario, every storage node may be evaluating a totally different set of operations and as a result the throughput of the system will be zero, even though all storage nodes are operation at full capacity.

In summary, simply saturating the system with queries will not give us useful information on how the system performs. Instead, we used a linearly increasing read-only workload and monitored the achieved throughput for different setups. We expect that as the workload rate increases, we will reach the limits of the system and beyond this point, we expect performance to deteriorate due to thrashing on the network.

Scale out

On the first scalability benchmark we try to find out how the system scales out in the number of storage nodes. In order to avoid measuring the effect of

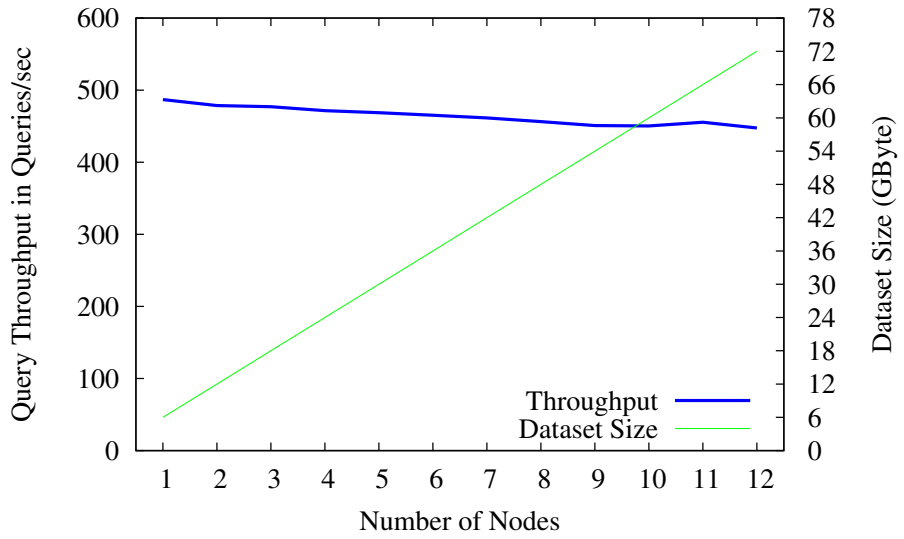


Figure 6.1: **Scale Out of the Distributed System**
Vary Number of Nodes, 6 GB Data per Node, Round-Robin Partitioning

skewed load balancing, we use round robin partitioning, both in our aggregation nodes and as our storage node segmentation strategy. Additionally, to avoid measuring the effect of network congestion, we used aggregation queries which return a single tuple.

Figure 6.1 illustrates the results of the experiment. The throughput curve represents the limit of the system. The slight performance decrease is due to the fact that having more storage nodes increases the probability that nodes execute different sets of queries. Note that since we are using round-robin partitioning, every incoming query has to be executed by every storage node in the system. This means that a locally shedding q queries results in $number\ of\ nodes * q$ less throughput. In a real deployment where hash partitioning is used, most queries affect a single storage node, thus the effect of load shedding is much smaller.

From the figure, it is obvious that as the dataset's size is increased while providing analogously more cores, the system exhibits almost the same performance. Thus we can conclude that the system scales out almost linearly.

Scale up

The second experiment focuses on how our distributed system, Daedalus, scales in the number of CPU cores per node. For this reason, we used all 12 storage nodes and increased the number of segments in each node from 1 to 6. As with the previous experiment, we maintained a constant segment size in all runs. We used again round-robin partitioning. That means increasing dataset size while adding more processing power. Under these circumstances, linear scale up means getting the same performance in each run.

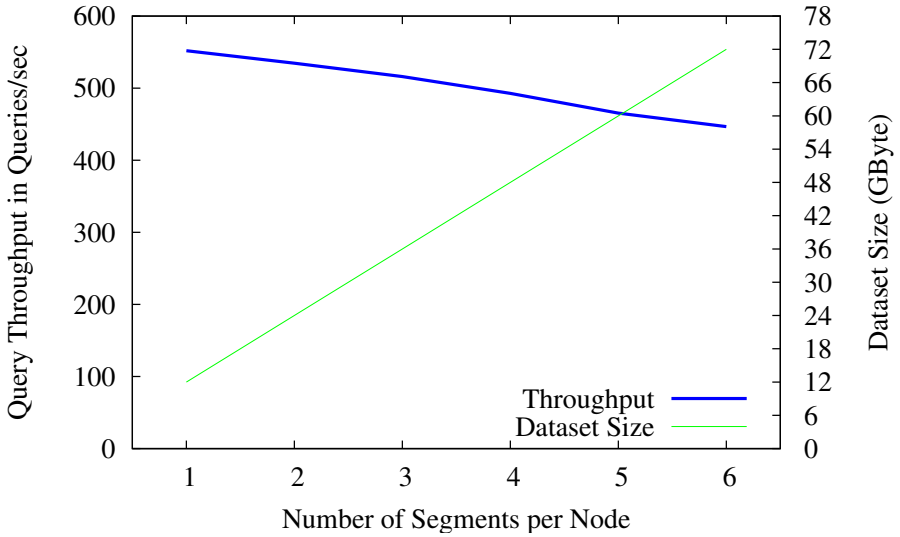


Figure 6.2: **Scale Up of the Distributed System**
Vary Number of Segments per Node, 1024 MB Data per Segment, 12 Nodes, Round-Robin Partitioning

The results are presented in figure 6.2. The result is consistent with the mathematical model presented in Section 2.3. To be precise, the effect of adding more memory segments to Daedalus is the same as the one presented in figure 2.13.

The performance deterioration that is observed as we increase the number of segments per node is only visible when we reach the limits of our system. For lower query rates, the queues that exist in the system are less full and as result the effect of desynchronization does not appear. All said, it is clear that our distributed system scales up in an almost linear manner.

Scale up vs scale out

The last scalability experiment compares the effects of scale up and scale out. This allows us to distinguish if it is better to scale *up* by putting more CPU cores into every node or to scale *out* by adding more smaller (and hence cheaper) machines to the system.

We used two different setups. The first one consists of only 2 storage nodes, each one having 6 memory segments storing 1 GB of data each. The total dataset that this setup holds is 12 GByte. In the second setup, we used 6 storage nodes, each one having 2 memory segments also storing 1 GB each. Again, the total dataset is 12 GBytes. As with the previous experiments, we measured the peak throughput of the system before performance starts deteriorating.

Figure 6.3 presents the results. It appears that adding more machines to the distributed system results in slightly higher performance than adding more

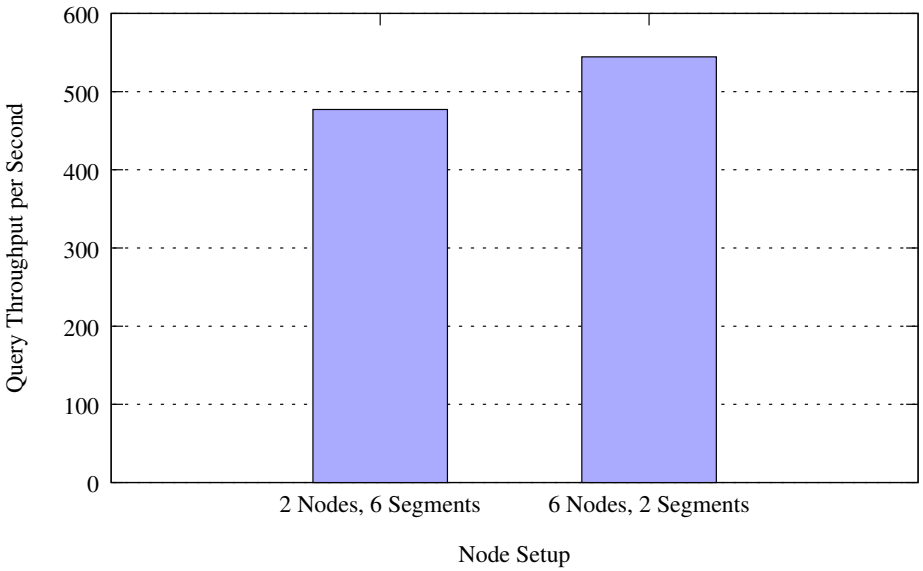


Figure 6.3: **Scaling of the Distributed System**
6 GBytes Total Dataset, Round-Robin Partitioning

processing power on each machine. The difference between the two cases is mainly due to the fact that the cumulative queue size in the second setup is bigger than the one in the first setup. Each node has its own incoming network buffers. This means that in the second setup, we have 3 times bigger queue than in the second case.

Of course, the performance gain comes at some price, which in this case is the CPU cores that are used for management of the storage nodes. Taking into consideration the current implementation which utilizes 2 CPU cores for management issues, the first setup requires 16 CPU cores while the second setup requires 24 CPU cores. For future implementations where it will be possible to bind the two management threads into the same CPU core, these numbers are 14 and 18 respectively.

The outcome of the scalability experiments is that the system scales slightly better as we add more storage nodes to the system, in terms of peak throughput. Scale up is also linear, but the gain is lower than the one of scale out.

6.4.2 System Limits

The second set of experiments is trying to explore the limits of our implemented distributed system. Scalability experiments show that the system scales both horizontally and vertically, however, as query rate and network traffic is increasing, we expect network congestion followed by packet and connectivity loss. Additionally, we want to identify the bottleneck of our system and what possible optimizations could overcome this bottleneck.

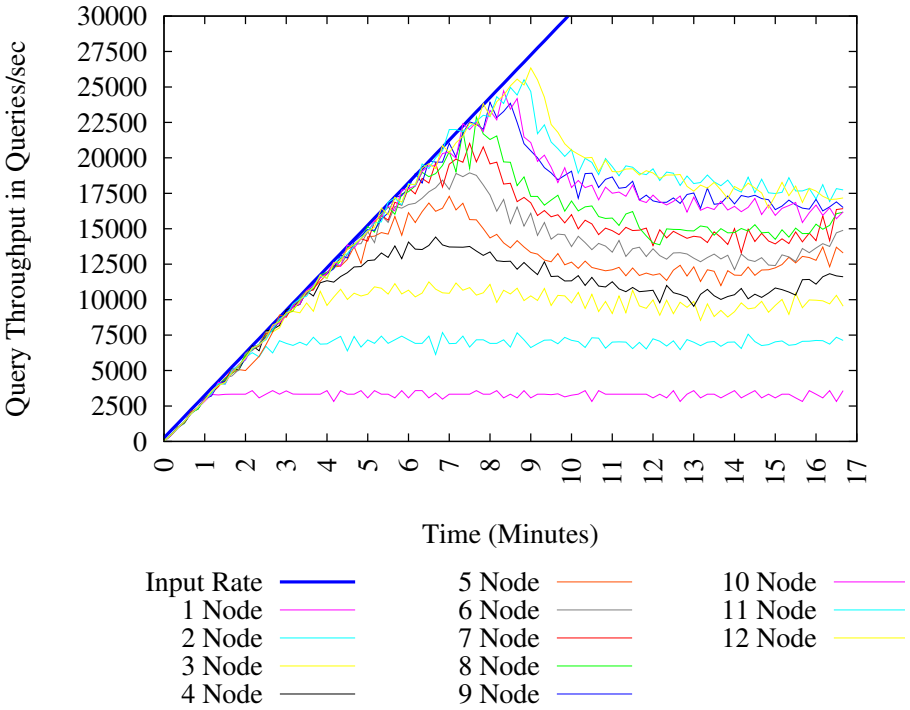


Figure 6.4: **Limits of Scaling of the Distributed System**
Vary Number of Nodes and Query Rate, Hash Partitioning, Monitor Throughput per Second

In this set of experiments, we used hash partitioning on both the aggregator and the controller layers as we increased the number of nodes that participated in the system. This means that the size of the dataset grows, however each node deals with only a constant-sized part of the dataset. We used `DATE_IN` as the partitioning key for the aggregation layer and `PRODUCT_ID` as the partitioning key for the Controller.

First, we used a variable query rate and monitored the throughput of the distributed system for setups with different numbers of nodes. The query rate was gradually increased by 50 queries per second. We used only one remote client and one aggregator. The throughput results of this experiment are presented in figure 6.4. Figure 6.5 shows the 50th, 90th and 99th percentile latencies per second while running the experiment with 12 storage nodes.

Our first observation is that for one storage node, pushing more queries than the system can handle does not hurt performance as the extra load is shedded. For more nodes, the effect of node desynchronization causes performance deterioration, as already explained. Additionally, we see that the system scales super linear in terms of throughput as we add nodes to the system. However scaling is not quadratic, as figure 6.6 indicates. Figure 6.6 shows the peak throughput achieved in each run.

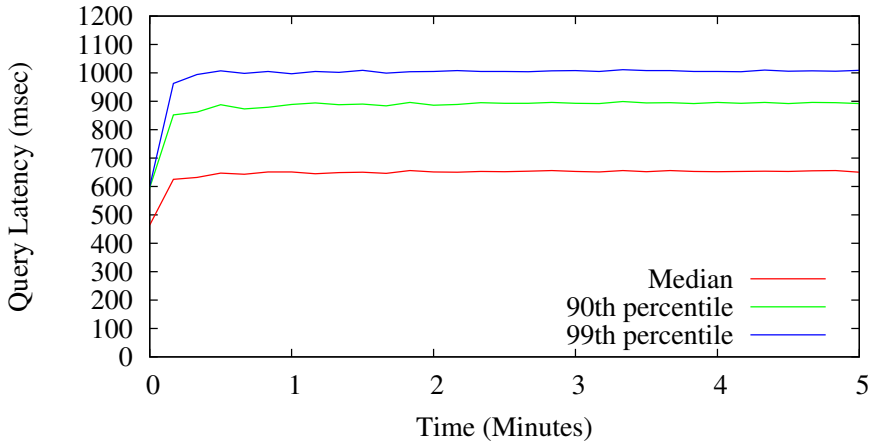


Figure 6.5: **Latency of the Distributed System**
12 Storage Nodes, Hash Partitioning, Monitor Latency per Second

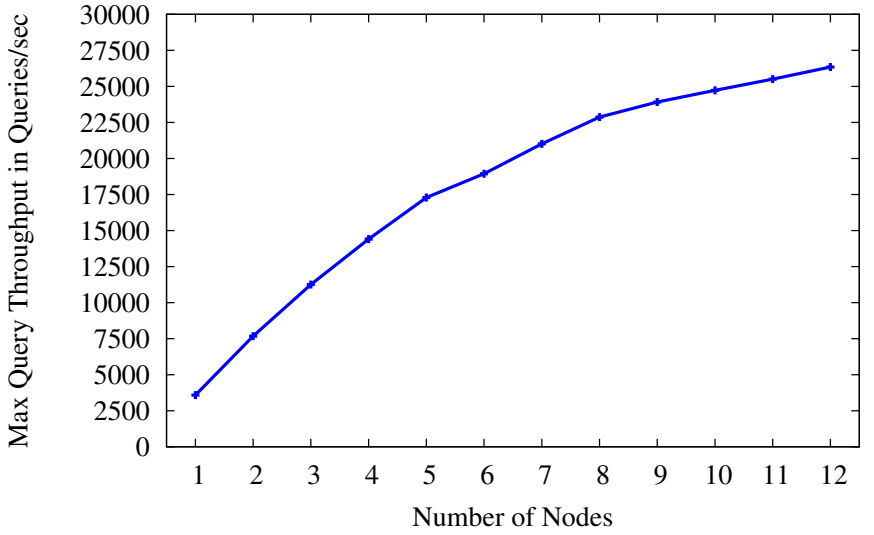


Figure 6.6: **Limits of Scaling of the Distributed System**
Vary Number of Nodes, Hash Partitioning, Monitor Max Throughput

At this point, one should mention that the implementation of the remote client client does not examine the whole incoming messages and thus the computational cost per incoming message is lower than the one in the aggregator or the storage node. In all experiments, when we mention query rate, we actually measure the number of operations that are sent by the client and not the expected query rate. As a result, the remote client cannot be the bottleneck of our system.

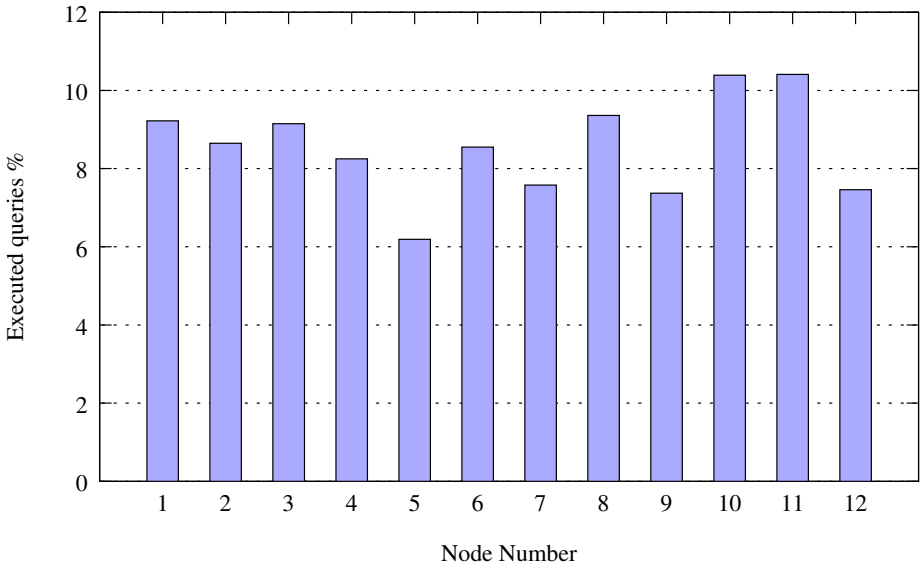


Figure 6.7: **Query load per storage node**
12 nodes, hash partitioning, monitor number of executed queries

We searched for an explanation for the sub-quadratic scale out in three different areas:

- **Load Imbalance**
- **Workload**
- **Aggregator capacity**

Load Imbalance

In our analysis of the Amadeus workload and dataset, we mentioned that the distribution of the hash values is not uniform as shown in figures 4.6 and 4.7. We expect this skew to exist in the distributed system.

We repeated the experiment of figure 6.4 with 12 nodes, but this time we measured the number of executed queries per storage node. The distribution of the query load to storage nodes is illustrated in figure 6.7.

From the figure, it is clear that not all storage nodes are operating at their full capacity, which of course results in degraded performance. There are certain ways to mitigate this problem, namely better hash functions and replication. These techniques are outside of the scope of this thesis.

Workload

Section 4 presented our throughout analysis of the Amadeus Workload. At the time of writing, most of the queries of the amadeus workload select on the key of the existing database, the `PRODUCT_ID`. However, a small percentage of them

(about 0.22%) do not have an equality predicate on the key. Such queries have to be executed by every single storage node. In our mathematical model, we modeled such queries by giving them a weight equal to the number of partitions, while point queries have a weight of 1. This technique can be applied also here.

A closer examination of figure 6.7 reveals that the sum of the load of all nodes is not 100% as expected, but 102.6%. This confirms our expectation, that a load of 99.78% queries selecting on the hash attribute and 0.22% queries not selecting on it, will inflict an additional $0.22 * 12$ cumulative load to the storage nodes. As we increase the number of partitions in our system, the queries that do not match the partitioning key dominate the load, leading to linear scale out as with round-robin partitioning.

Aggregator Capacity

Load imbalance and workload composition explain the performance drop that was observed in figure 6.6. However, we still need to measure the capacity (in queries per second) of the aggregator nodes to determine if they could become a bottleneck in the system. In order to find the aggregator limitations, we had to perform certain modifications to our system.

First of all, we created a "void" storage node, exposing the same RPC interface to the aggregator as the normal storage node. The void storage nodes creates instantly a single end-of-operation result for every incoming operation, without storing any data or performing any scan. We measured the performance of the void storage engine on a system with no aggregators and a client node connected directly to the void storage node. The throughput we measured was more than 85,000 queries per second.

Next, we repeated the same experiment with the introduction of an aggregator node between the client and the storage layer, consisting of 12 void storage nodes. As before, we used an increasing query rate and monitored the throughput of the system.

The performance of the "void distributed system" is shown in figure 6.8. It appears that the limit of the aggregator is slightly lower than the limit of the system without the aggregator node: 84,000 queries per second versus 85,000 queries per second respectively. We can safely assume that 85,000 is the maximum capacity of the RPC library and that the 1,000 queries decrease in throughput is inflicted by the aggregator logic.

While performing this experiment we also measured the latency of queries per second to create figure 6.9. This figure shows that the aggregator node's overhead is less than 30 msec for 99 percent of the queries. The 99th percentile spikes that are observed at the beginning of the experiment are caused by some random broken TCP connections we encountered while executing the experiment. However, even if a TCP connection breaks, the aggregator node reconnects and continues streaming results with less than 30 millisecond delay.

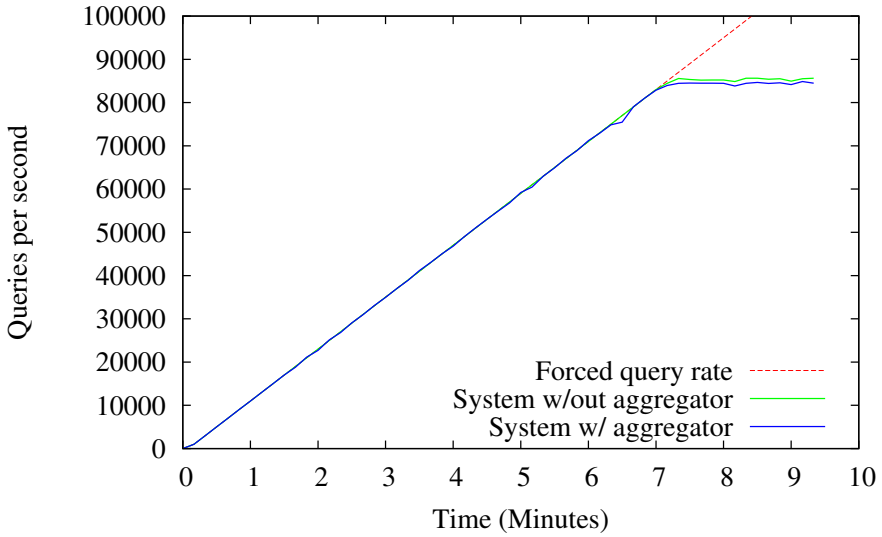


Figure 6.8: **Void Storage Engine Performance**

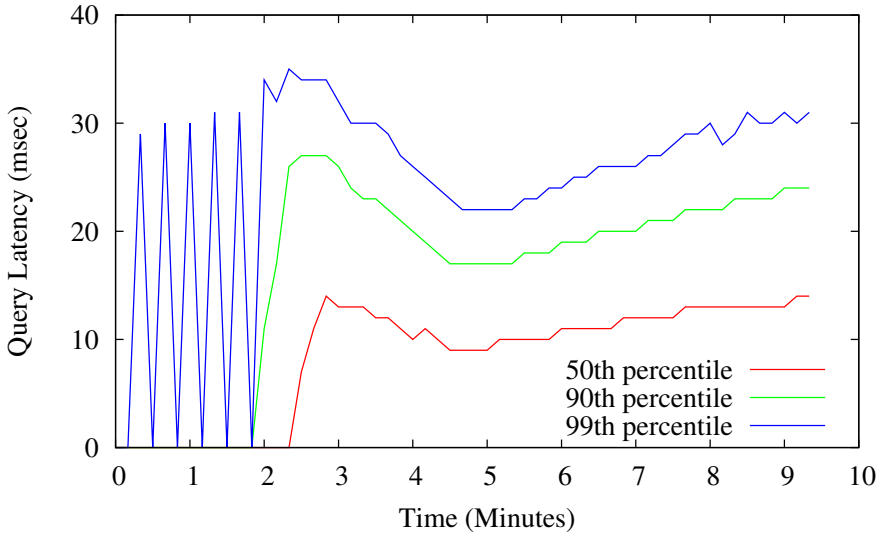


Figure 6.9: **Void Storage Engine Query Latency**
Latency per second of execution while applying increasing query rate

The outcome is that one aggregator is sufficient to cope with the load of our distributed system. The performance drop that was observed in figure 6.6 as we added more cores is an artifact of the Amadeus workload composition and skew of our attribute values.

6.4.3 Updates

In Chapter 5 we explained why updates have to be sent to every node. Additionally, since updates are transmitted via UDP and Daedalus does not implement selective repeat, a lost update message may cause a large number of re-transmissions and thereby *additional* network load. In this experiment we try to measure the way updates scale in Daedalus.

We used a variable number of storage nodes, each one containing 6 memory segments storing 1 GByte each, giving a cumulative dataset of 6 GB per storage node. Each node has a limit of 256 on the size of the running operation queue in order to meet the 2 second latency requirement. For easier comparison and consistence with the previous experiments, we again segmented data across segments using `PRODUCT_ID` as our hashing key. Segmentation on `PRODUCT_ID` provides no benefits for updates, since all of them select on `RLOC` as illustrated in figures 4.3 and 4.4. However, according to our analysis in Section 4.1, this setup results in better load distribution.

The setup contains a single aggregator node, as the experiment of Section 6.4.2 showed that the capacity of one aggregator is much higher than the throughput of our distributed system. However, in this case, since we are interested in the scalability of updates, we used `RLOC` as the hashing key. The final system is expected to perform multi-level hash partitioning on both `RLOC` and `PROVIDER`, thus partitioning both update and query workload across storage nodes.

We used an increasing update rate starting at 0 updates per second and linearly increasing by 20 updates every second, and monitored the throughput and latency of the system. The results are presented in figures 6.10 and 6.11. Figure 6.10 illustrates the throughput of the distributed system. As expected, pushing the system beyond its capacity leads to heavy thrashing. The extra updates are disposed, as they cannot fit into the queue and then the storage nodes have to reject them. The rejection propagates back to the client which has to re-transmit the update and everything that followed.

We found that the median round trip time in our network is around 10 milliseconds (*cf.* figure 6.9). During these 10 milliseconds, the storage nodes is not allowed to handle any updates in order to maintain write monotonicity. As a result, throughput quickly decreases to a value very close to 100 (1 second over 10 milliseconds).

Figure 6.11 illustrates the peak throughputs for all 12 setups. From the curve, it is obvious that the system does not scale linearly in the number of nodes. This is explained by the fact that the aggregator "broadcasts" all updates to the storage nodes. Nodes that should be involved in evaluating the update receive the full update message, while all other nodes receive a NO-OP operation. Since broadcasting is performed using unreliable communication (UDP datagrams), the probability of successfully delivering an update to every node is exponentially decreasing . As already mentioned, in case of update message loss, the client has to reissue the update, which requires 10 extra milliseconds.

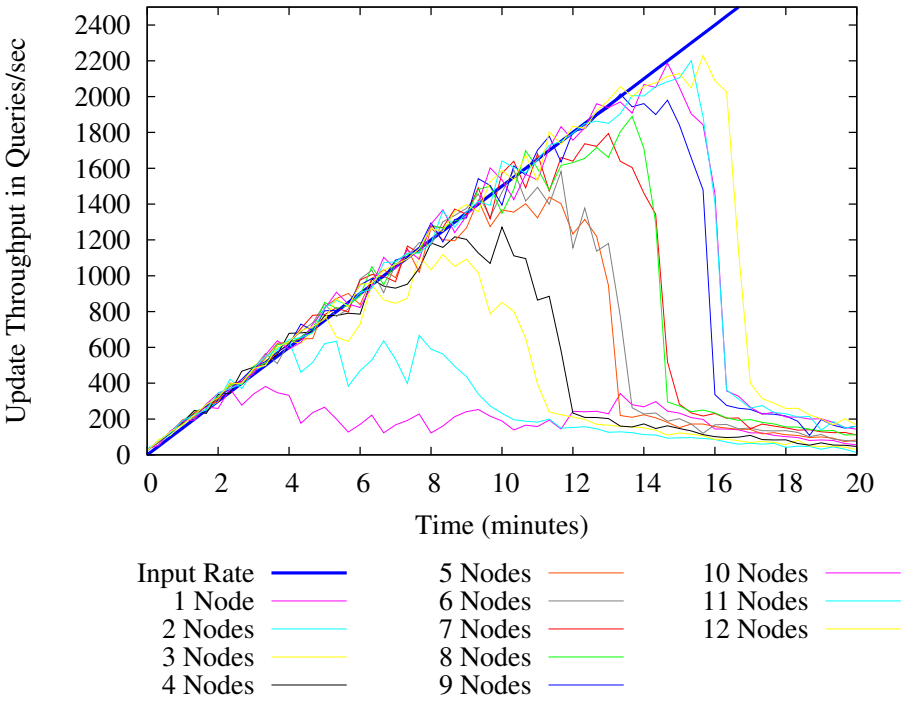


Figure 6.10: **Update Scalability**
Update Throughput, Vary Number of Nodes, Partition on `RLOC`

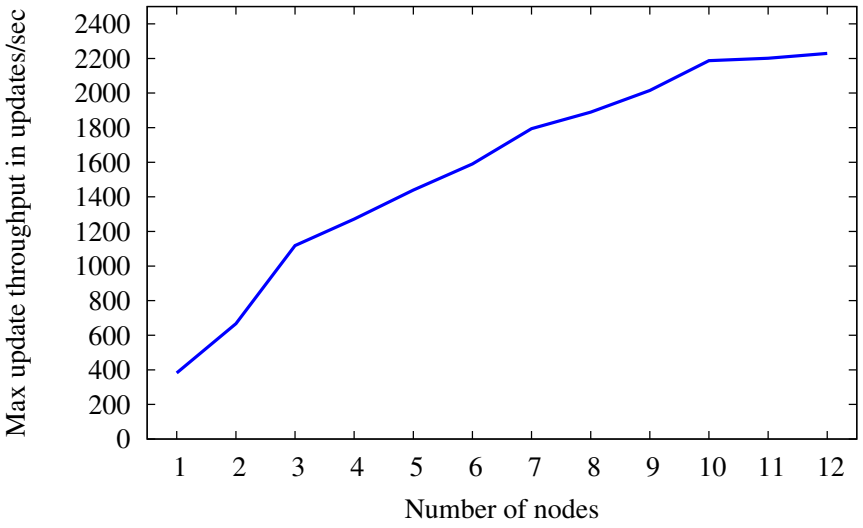


Figure 6.11: **Update Scalability**
Max Update Throughput, Vary Number of Nodes, Partition on `RLOC`

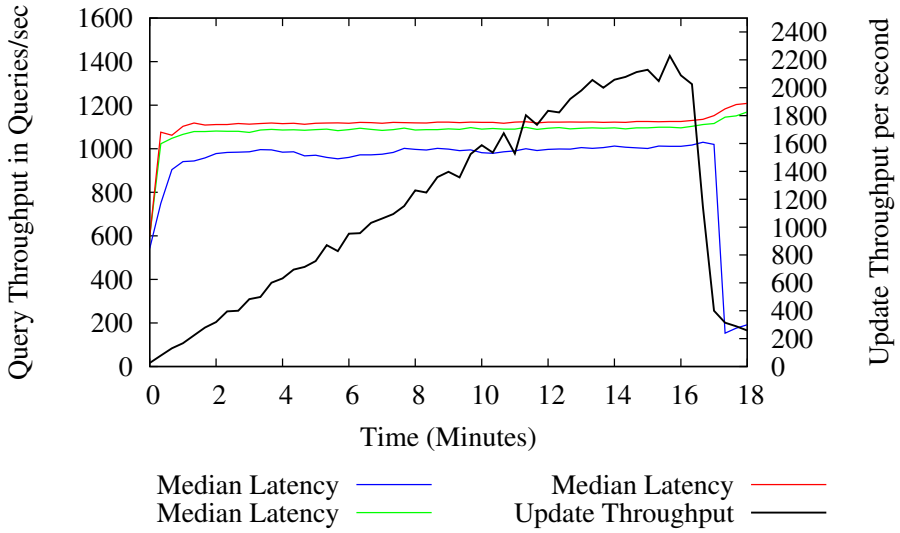


Figure 6.12: **Update Latency**
12 storage nodes, 6 GB each

Part of the sub-quadratic scale up is attributed to load imbalance. The way load balancing affects performance of queries was analyzed in Section 6.4.2. Updates behave analogously.

Finally, figure 6.12 depicts the median, 90th and 99th percentile latencies of Daedalus with 12 storage nodes. We can see that every update is applied in less than 1.2 seconds which meets our latency requirement of 2 seconds. More importantly, latency remains constant as the system is scaled out.

6.4.4 Impact of selectivity

In the last experiment, we tried to measure the effect of the query selectivity on our system. As queries select more and more tuples, we expect both the aggregator and the storage node performances to drop dramatically. For this experiment we used read-only workload of an increasing rate by 50 queries per second. As with all read-only experiments, we used a single aggregator node partitioning on `DATE_IN`.

In order to control the results generation, we extended the "void" storage engine we presented in Section 6.4.2 with minor modifications. The updated "void" storage engine generates a predefined number of results for every query and then issues the special *end-of-results* message. Each result contains a full record, which is 322 bytes and not a projection of it.

The results are presented in figure 6.13. The figure shows that as the number of results increases, the achieved throughput drops exponentially. However in all cases the product of the number of results times the throughput, is around

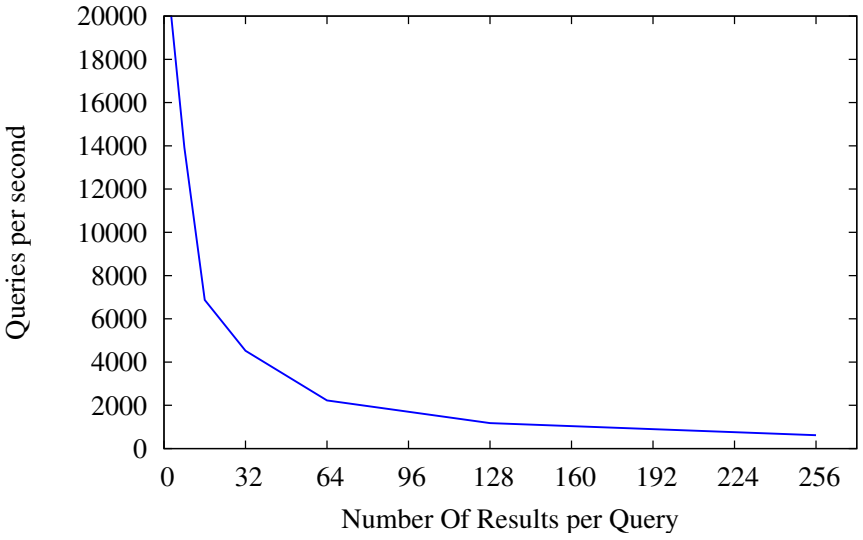


Figure 6.13: **Impact of Selectivity**
12 "Void" Storage Nodes, 1 Aggregator, Vary Number of Results per Query

84,000. In other words, the maximum capacity of the aggregator in processed messages per second is around 84,000 messages per second which supports our assumptions of Section 6.4.2.

In order to support higher query and result rates, we can use more aggregator nodes. Additionally, we can extend the implementation of the RPC library to allow sending multiple queries and results in one network packet. This optimization is our of the scope of this thesis and considered future work.

Chapter 7

Conclusions

During this thesis, a simple yet fully operative distributed storage system was designed and implemented. The system, named "Daedalus", is based on nodes running the Crescando Storage Engine which provides predictable performance for unpredictable workload, as supported by empirical results and a mathematical model that was introduced in this thesis. The system was designed to be used as a materialized view on the Global Distribution System (GDS) that Amadeus SA [1] is using to store airline tickets.

The distributed system is designed to scale up in the number of CPU cores and scale out in the number of storage nodes while offering atomicity, write-monotonicity and durability guarantees. Additionally the system provides predictable response time for any workload, given an incoming query rate. Finally, the system is very flexible to the diversity of the queries and the update load it is able to handle.

The implementation includes:

- a host application for the clock scan library,
- a library for sending messages over unreliable networks with integrity guarantees,
- a library for serializing remote operations and results,
- a query-routing application that serves as the middleware layer between storage nodes and clients and
- a client-side library

The implemented system was based on a number of assumptions. All nodes are considered static (no node membership issues) and recovery can only be performed by bringing the system offline. Even under these assumptions, the distributed system is fully operational and represents a distributed system under normal operation (when lack of failures).

We have evaluated the distributed system using real data from the Global Distributed System and found that it matched our expectations. We observed an almost linear scale out in the number of storage nodes used. The overhead the distributed system introduces is minimal ranging from almost 0 milliseconds to just 40 milliseconds, based on the load of the system.

7.1 Future Work

7.1.1 Replication

Currently, there is no notion of replication in Daedalus. The dataset is stored in the main memory of a set of machines (storage nodes) and checkpointed to persistent storage (hard disks) as explained in Section 2.1.3. However, in case of a machine failure, the system has to be taken offline until the storage node recovers and the dataset is copied from the persistent storage to main memory.

In order to guarantee six-nine availability in Daedalus, we need to overcome the issue of single node failures. For this reason, we intend to implement replication at the storage layer. The first step would be to create groups of nodes that share the same partition of the dataset. In such a "replication group", one node is responsible for managing the other nodes. This "master node" accepts all operations targeted to this dataset partition and routes them to the other nodes.

7.1.2 Distributed Recovery

Distributed recovery requires that a storage node is able to re-join a replication group. In order to maintain a consistent dataset, the master node is in charge of assigning nodes to assist the new node in retrieving the dataset.

Since every storage node stores a few gigabytes of data, we consider using RDMA[4] which allows data to move directly from the memory of one machine into that of another without involving either one's operating system. This permits high-throughput and low-latency networking.

7.2 Node Membership

As already mentioned, the current implementation assumes that nodes are static. No new nodes can be added in either the aggregation or the storage layers. We plan on using a set of special nodes to address this node membership problem. These well known, fault tolerant nodes will operate as a "catalog" of the nodes of the system.

7.3 Dynamic Partitioning

Dynamic partitioning is an alternative approach of dealing with the load balancing dimension of our problem. The replication group that experiences the higher load can be split in two, thus reducing the load. Dynamic partitioning

has not been analyzed yet. A promising approach is usage of distributed hash tables (DHT).

Appendix A

Message Layouts

crescando
Network Protocol Specification
Message Layouts

Version	Last Author	Date
0.1	Philipp Unterbrunner	12.12.2008
0.2	Georgios Giannikis	26.03.2009
0.3	Georgios Giannikis	14.04.2009

A.1 Document Scope and Content

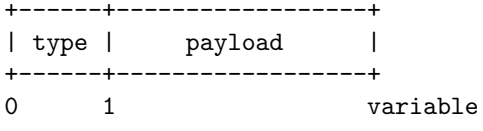
This document specifies the physical layout of crescando application messages passed over the network. By message, we mean byte-arrays of arbitrary but finite length passed between crescando nodes as seen by the application layer. The mapping of these byte-arrays to UDP or TCP datagrams is defined by the message layer and specified in a separate document.

A.2 General Specifications

All integer types are little-endian. Offsets and sizes of data structures are in bytes unless otherwise indicated. A byte is 8 bit.

Any fault not explicitly covered by this document (e.g. an invalid type value) is to be considered a critical implementation error whose effects are undefined.

A.3 General Layout



The first byte in a message is an unsigned 8-bit integer, which specifies the type of the message according to which the payload is interpreted. Type values map to names as follows:

Type Value	Type Name
0	null (keep alive message)
1	enqueue_select
2	enqueue_aggreagate_select
3	abort_select
5	enqueue_delete
6	enqueue_update
7	enqueue_insert
8	enqueue_no_op
9	select_result_arrived
10	write_confirmed
11	write_rejected

A.4 Generic Types

Predicate List

A predicate list is used in various operations. The serialized size of a predicate list is variable and the payload layout is as follows:

```
+----+------------+-----------+--------------+
|size| attributes | operators | value buffer |
+----+------------+-----------+--------------+
0    2            var         var            var
```

Attribute	Description
size	a two-byte integer which represents the number of predicates that the list contains
attributes	the size of this field is #size bytes. It contains the attributes used in the predicates vectorized. Every attribute is represented by a 4 byte integer
operators	the size of this field is #size bytes. It contains the operators used in the predicates vectorized. Every operator is represented by a 4 byte integer
value buffer	the predicate values, vectorized. The size of each value is schema dependent. The deserializer should be able to calculate the size of each predicate value by examining the attributes field.

The size of a predicate list is 2 + 2 * 4 * listSize + valuesBufferSize.

Attribute List

An attribute list is used in various operations. The serialized size of an attribute list is variable and the payload layout is as follows:

```
+----+------------+
|size| attributes |
+----+------------+
0    2            var
```

Attribute	Description
size	a two-byte integer which represents the number of attributes that the list contains
attributes	the attributes of the list, vectorized. The attributes are represented as a 4 byte integer

The size of a predicate list is 2 + 4 * listSize.

Values List

A value list is used in update operations. The serialized size of a value list is variable and the payload layout is as follows:

```
+--------+
| values |
+--------+
0        var
```

A value list depends on an attribute list. The deserializer should have knowledge of the schema and the attribute list in order to deserialize a value list. The size field represents the size of the values field. The values field is the values vectorized.

A.5 Payload Layout

null

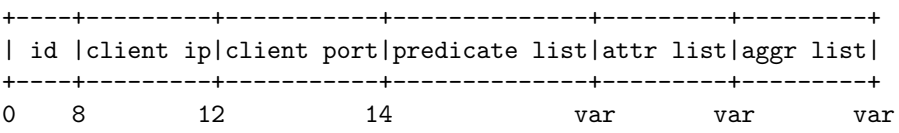
Null messages have no payload. Their sole purpose is to keep TCP connections alive (or detect that they have broken).

enqueue_select

```
+--+---------+-----------+--------------+--------------+
|id|client ip|client port|predicate list|attribute list|
+--+---------+-----------+--------------+--------------+
0  8         12          14             var.           var.
```

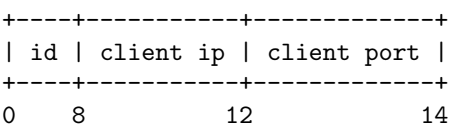
Attribute	Description
id	The operation id of this operation
client ip/client port	Define the endpoint that the results of the select should be forwarded to.
predicate list	A generic Predicate List containing the predicates of this operation
attribute list	A generic Attribute List containing the projected attributes

enqueue_aggregate_select



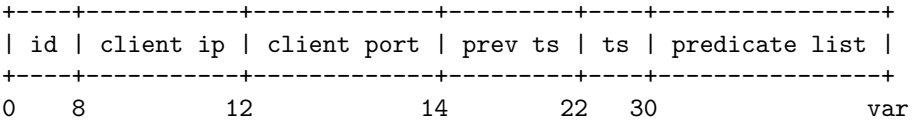
Attribute	Description
id	The operation id of this operation
client ip/client port	Define the endpoint that the results of the select should be forwarded to.
predicate list	A generic Predicate List containing the predicates of this operation
attr list	A generic Attribute List containing the projected attributes
aggr list	A generic attribute list containing the aggregate functions

enqueue_abort_select



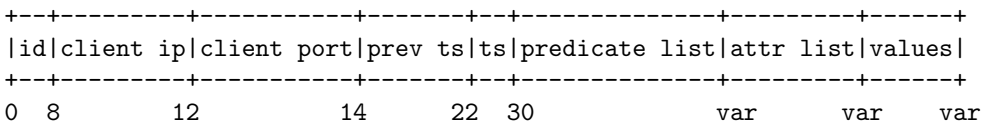
Attribute	Description
id	The operation id of this operation
client ip/client port	Define the endpoint that the results of the select should be

enqueue_delete



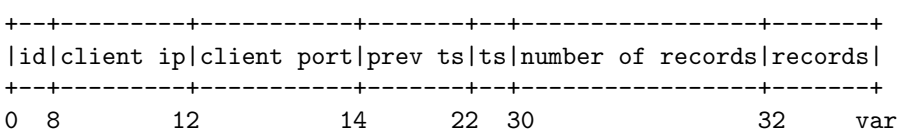
Attribute	Description
id	The operation id of this operation
client ip/client port	Define the endpoint that the results of the delete should be forwarded to
prev ts/ts	The previous timestamp/current timestamp, represented as 8 byte integers
predicate list	A generic Predicate List containing the predicates of this operation

enqueue_update



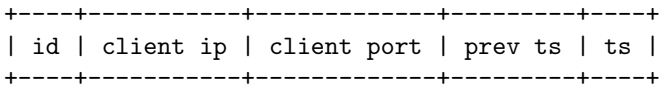
Attribute	Description
id	The operation id of this operation
client ip/client port	Define the endpoint that the results of the update should be forwarded to.
prev ts/ts	The previous timestamp/current timestamp, represented as 8 byte integers
predicate list	A generic Predicate List containing the predicates of this operation
attr list	A generic Attribute List containing the attributes that will be updated
values	A generic Values List containing the new values for the attributes

enqueue_insert



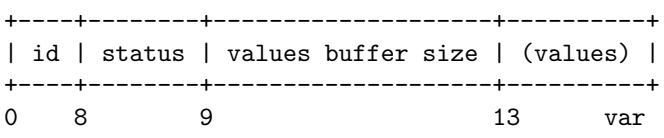
Attribute	Description
id	The operation id of this operation
client ip/client port	Define the endpoint that the results of the insert should be forwarded to
prev ts/ts	The previous timestamp/current timestamp, represented as 8 byte integers
number of records	A 2-byte integer representing the number of records this insert contains
records	The records of this operation, serialized

enqueue_no_op



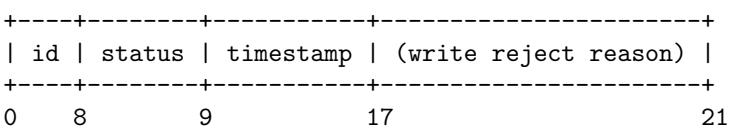
Attribute	Description
id	The operation id of this operation
client ip/client port	Define the endpoint that the results of the no-op should be forwarded to
prev ts/ts	The previous timestamp/current timestamp, represented as 8 byte integers

select_result_arrived



Attribute	Description
id	The operation id of this operation
status	A character representing the status of the result. Possible status values: 0 Select is in progress 1 Select has completed
values buffer size	If the status of the result is 0 (select in progress), this 4-byte integer holds the size of the values field. If the status of the result is 1 (select completed), this field contains the value 0 (as a 4-byte integer)
values	The values of the select result. This field is only present if the status of the result is equal to 0 (select in progress)

write_result_arrived



Attribute	Description
id	The operation id of this operation
status	A character representing the status of the result. Possible status values: 2 Write was executed 3 Write was rejected
timestamp	The timestamp of the given operation
write reject reason	If the status of the result is 1 (write operation rejected), this 4 byte integer holds an identifier for the rejected reason: 0 Could not enqueue, because queues were full 1 Bad timestamp. In this case the timestamp field would contain the expected timestamp.

Bibliography

[1] Amadeus SA. URL http://www.amadeus.com.

[2] Advanced Micro Devices. Quad-Core AMD Opteron processors for servers and workstations. AMD Whitepaper, 2007. URL http://www.amd.com/us-en/assets/content_type/DownloadableAssets/FINAL_AMD_Opteron_Overview_43753A.pdf.

[3] European Computer Manufacturers Association. Standard ECMA-182, Data Interchange on 12,7 mm 48-Track Magnetic Tape Cartridges - DLT1 Format. ECMA Whitepaper, 1992. URL http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-182.pdf.

[4] Philip Frey and Gustavo Alonso. Minimizing the Hidden Cost of RDMA. ICDCS 2009.

[5] Andi Kleen. A NUMA API for Linux. Novell Whitepaper, 2005. URL http://www.halobates.de/numaapi3.pdf.

[6] P. Unterbrunner, G. Giannikis, G. Alonso, D. Fauser, and D. Kossmann. Predictable Performance for Unpredictable Workloads.

[7] P. Unterbrunner, G. Giannikis, G. Alonso, D. Fauser, and D. Kossmann. The crescando Storage Engine. Technical report, Systems Group, Dept. of Computer Science, ETH Zurich, Switzerland, 2008.