Loading and querying data on distributed virtualized web application servers

Moritz Mack
Abstract

Loading and querying data on distributed virtualized web application servers

Moritz Mack

Virtualized web application servers within computational clouds, such as the Google App Engine, generally restrict resource usage and therefore provide limited, relationally none-complete query facilities only. This work investigates how scalable, reliable and a more powerful access to the App Engine Datastore can be accomplished and an Optimized Distributed Datastore Exchange (ODDSE) is presented. Being aware of the App Engine’s resource restrictions ODDSE provides a reliable and failure safe query interface to transparently exchange data with the distributed Datastore using full SQL or AmosQL. ODDSE therefore wraps Datastore relations and utilizes the extensible database system Amos II to compensate for missing query facilities in Google’s relationally none-complete query language GQL. Under the covers ODDSE furthermore implements an adaptive and reliable management of interactions with App Engine servers. For scalability and high performance the interaction is parallelized and continuously adapted. The performance of ODDSE is evaluated and compared to a similar system showing its considerably good results for both bulk uploading and query processing.
Contents

Abstract i

Contents iii

1. Introduction 1

2. Background 6
   2.1. Database technology ............................................. 6
       2.1.1. Relational Database Management Systems ................... 6
       2.1.2. Distributed storage system - Bigtable .................... 8
   2.2. Amos II .......................................................... 9
   2.3. Cloud Computing and App Engine ................................ 11
       2.3.1. Google App Engine ......................................... 13

3. ODDSE (Optimized Distributed DataStore Exchange) 17
   3.1. System overview .................................................. 17
   3.2. Supported Scenarios ........................................... 23
       3.2.1. Schema integration and exchange ........................... 23
       3.2.2. Query processing ........................................... 25
       3.2.3. Bulk loading of data ...................................... 27
   3.3. Architecture ..................................................... 29
       3.3.1. Metaschema ................................................... 29
       3.3.2. The ODDSE communication protocol ......................... 35
       3.3.3. The ODDSE client ............................................ 39
       3.3.4. The ODDSE server .......................................... 49

4. Bulk loading 56
   4.1. Chunk generation ............................................... 60
       4.1.1. Sequential approach ....................................... 60
1. Introduction

In the panel description of the 24th International Conference on Data Engineering (IEEE) the author brings up Thomas Watson’s well-known - even though unverified - prediction and speculates if Watson, in the end, was right after all. Supposedly having said “I think there is a world market for maybe five computers” in 1943, Thomas Watson might have been years ahead of his time. Taking the testimony more general, thinking rather about farms of computers than single computers, we can actually notice some concordance to a current paradigm shift [Ram08]. Companies start moving part of their IT infrastructure from traditional self-owned data centers into a few centric virtualized next-generation data centers offering infrastructure as a service [BYV+08]. A popular example for Infrastructure as a Service (IaaS) is Amazon’s Elastic Compute Cloud (EC2).

Outsourcing IT infrastructure to computational clouds, providing pay-as-you-go resources, not only allows companies to reduce their investments in IT. They can moreover focus on their key competencies spending less time and effort on maintaining and administrating their own IT infrastructure. Besides they gain flexibility to adapt their - virtual - infrastructure from one moment to another as changing demands require[BYV+08, Gee09].

With the Google App Engine Google launched its own cloud service in April 2008 leading to this thesis project. While the focus of Amazon’s EC2 is on infrastructure, the App Engine offers a restrictive platform in order to promote scalable web applications [Sla08]. Google is currently offering this service for free within certain resource usage quotas. Additional quotas can also be purchased.

Enabling nearly everybody to easily build scalable applications even without pre-knowledge of parallelization on top of a more or less free service, the App Engine is most likely to take off and become yet another Google success. Nevertheless, following relevant blogs [O’G08, O’R08] and user groups [Dis08] on the topic, there are also doubts about the App Engine and its usability. The concern addressed most frequently describes the lock-in issue that binds your hopefully prosperous business idea for ever to Google’s infrastructure
1. Introduction

Figure 1.1.: The above chart illustrates the fast growing web search popularity of cloud computing in comparison to related paradigms such as grid computing and others. It is based on data obtained from Google Trends measuring current search trends. Following [WVLKT08] the success of cloud computing resides in the provision of simple user-centric interfaces while offering great flexibility and scalability.

making it their fortune instead of yours. This particularly becomes obvious in terms of a lock-in of business data and is discussed by [AFG+09] as the cloud computing obstacle number two right after the availability of subscribed cloud services.

The obstacle of data lock-in is not only restricted to migration of infrastructure. Performing more complicated queries, e.g. data analysis for decision making, may be challenging because of limitations with the provided infrastructure. For example, App Engine does not allow to return result sets from database queries larger than 1,000 records.

Furthermore, the App Engine platform enforces restrictive timeouts and resource usage quotas on CPU cycles, on Datastore API calls to the back-end-Bigtable storage manager [CDG+06], as well as limits on the amount of received and sent data and much more. Due to these restrictions even reliability may be compromised. The reason for the limitations is to provide the necessary system scalability. For example, Google App Engine is built upon a highly scalable but restricted storage system called Bigtable [CDG+06]. While it enables Google App Engine to scale largely, developers need to rethink their concepts and
architectures.

In this thesis project it is investigated how scalable and reliable access to databases stored in Bigtable can be accomplished and how data can be up and downloaded efficiently to Bigtable databases.

The thesis focuses on a scenario in which a query processor running on some client computer receives user queries in SQL over relations managed by Bigtable through Google App Engine. The query engine sends query requests to collaborating App Engine servers and composes the query result. Each App Engine server is running special software in order to service the query clients.

The following questions are fundamental in this scenario and are hence thoroughly investigated in the project:

- How can Bigtable relations be utilized transparently in queries not having the restrictions imposed by Bigtable?
- How can scalable and reliable query processing be achieved utilizing the limited resources of App Engine applications?
- How can data be up and downloaded to the collaborating servers as simple and scalable as possible?

Answers to the above questions are realized in a fully functional implementation called ODDSE (Optimized Distributed DataStore Exchange). ODDSE utilizes the extensible database system Amos II [RJK04].

The system consists of ODDSE servers implemented in Python that run as App Engine applications. They service ODDSE clients, which are Java-based plug-ins to Amos II systems running in client computers. The ODDSE clients and servers communicate using a HTTP-based protocol.

ODDSE wraps Bigtable relations and provides transparent queries over them. By leveraging the Amos II query processor, the system is able to compensate for missing query facilities in Google’s relationally non-complete query language, GQL. ODDSE enables general queries expressed either in SQL or AmosQL [RJK04] on accessed Bigtable relations. By using the multi-database facilities of Amos II [RJK04], Bigtable relations can even be joined with data from various other data sources [Geb99].
1. Introduction

In addition to the provision of query interfaces, ODDSE provides an interface for bulk up and downloads of data into Bigtable relations. This enables backup of Bigtable relations in case of failures and provides data storage independence of App Engine.

Being furthermore capable of creating new Bigtable relations on App Engine servers ODDSE accomplishes high flexibility and enables the usage of Bigtable as a highly available cloud storage system being utilized from local ODDSE client computers.

Under the covers ODDSE implements an adaptive and reliable management of interactions with App Engine servers to access Bigtable relations. For scalability and high performance the interaction is parallelized and continuously adapted.

Reliability in ODDSE is based on fault-tolerant resumable query managers. A resumable query manager is a process servicing a query in a fault-tolerant way to overcome App Engine failures such as timeouts or quota violations.

To show the suitability of the proposed implementation and in particular of the proposed resumable query managers, ODDSE was evaluated with respect to performance (data throughput), scalability, and reliability. Evaluations were made both for bulk uploads as well as for query search performance. Improvements achieved using the resumable query managers are compared with naive approaches. The comparison to related work shows that ODDSE performs very well.

The remainder of this thesis is organized in the following way.

Chapter 2 gives the required background on the Google App Engine and Bigtable, in particular clarifying the earlier mentioned restrictions in detail. Additional information on cloud computing and database systems in general shall help to get a broad picture and showing how ODDSE fits in. The section continues by describing Amos II, which is utilized by ODDSE to compensate for Bigtable’s limited query facilities, giving the appropriate background to understand how ODDSE processes queries. Finally the chapter ends with a general description of cloud computing and the AppEngine.

Chapter 3 gives a system overview and illustrates the usage of ODDSE. The chapter presents the architectural concept of the system, its metadata as well as its communication protocol. Last but not least the implementation of the ODDSE client and server is thoroughly discussed.

Chapter 4 describes the bulk upload mechanism in ODDSE. Generation of data chunks
from bulk files is discussed as basis to enable scalable data upload to Bigtable relations by parallelism. Furthermore, the interaction optimizer that manages adaptive client-server interaction is described. Finally, the presented ODDSE bulk uploader is evaluated and compared to the App Engine’s own bulk loading tool.

Chapter 5 focuses on query processing in ODDSE. This chapter first describes how to partition user SQL queries into several chunk queries in GQL. The adaptive interaction optimizer for queries is presented. The performance of ODDSE query processing is finally evaluated in terms of data throughput, reliability, and scalability.

Related approaches to deal with limitations on the App Engine Bigtable and other related work is described in Chapter 6.

Chapter 7 finally gives a conclusion of the implemented system mentioning the main contributions given by ODDSE as well as outlining some future work.
2. Background

2.1. Database technology

Database systems aim to provide efficient and scalable processing of huge amounts of data. In particular when being exposed to the world wide web, scalability becomes difficult to achieve due to exploding data volumes and nearly unpredictable highly dynamic workloads.

In terms of database technology, structured data is stored in a database which is managed by a database management system (DBMS). Data in such databases is processed by specifying queries to DBMS in a specific query language [GMUW02], usually SQL.

In this thesis scalability of database technology is one essential focus. Hence, in the following a short introduction is given on technologies being referred to in this thesis and their scalability is discussed. The usage of the database system Amos II, used in the client side in ODDSE, is explained in the required manner.

In addition some different data models are introduced in order to give an understanding of the model transformations performed in the project.

2.1.1. Relational Database Management Systems

Relational DBMS (RDBMS) build up on the relational data model. Basic concepts in this model are mathematical relations (stored as tables by the RDBMS) and a family of operators on these relations such as, e.g., project \( \pi_{a_1, a_2, \ldots, a_i}(R) \), select \( \sigma_{\text{condition}}(R) \) or join \( (R_1 \bowtie R_2) \) [Cod70, SS75]. The relational algebra defines all basic, meaningful operations on relations. Relational query languages that are as least as powerful as relational algebra are called relationally complete. This applies for example to SQL, which is certainly the most well known query language and hence not explained further at this point.

The entity-relationship model (ERM) [Che76] is a high level approach to model data when...
designing a database. This design phase is generally referred to as **conceptual modeling** and generally presented in diagrams using the **entity-relationship** (ER) model. In ER **entity types** correspond to real-world objects. **Attributes** of entity types correspondingly model properties of these objects. Different entity types are interrelated by **relationships**. ERM can furthermore be extended with **inheritance** of entity types.

![ER Diagram](attachment:ER_Diagram.png)

**Figure 2.1.:** This figure presents an ER diagram following the notation of [Che76]. **Relationships** are defined together with their **cardinality**.

When transferring a conceptual model in ER notation into a database schema entity types are mapped to tables, and type attributes become columns of the corresponding table. Rows in relational tables refer to specific entities and are generally identified by a primary key. The reflection of an entity type relationship depends on its **cardinality**. So-called one-to-many (1:N and 1:1) relationships as shown in Figure 2.1 are realized including the primary key of the relationships unique side as foreign key column on the opponent side. Many-to-many (M:N) relationships however can’t be reflected this way requiring an additional relational table containing pairs of primary keys [Cod70].

The database provided by Google App Engine uses a data model very close to the ER model. In order to provide a relational view on data stored in App Engine ODDSE applies precisely the above presented transformations.

Achieving scalability with RDBMSs is an extraordinary difficult and expensive task even restricted to physical limits [GL02, CAA08]. The easiest way to increase scalability is certainly upgrading to state-of-the-art technology providing higher performance. RDBMSs however rely on hardware reliability generally making machines expensive. Redundancy and replication of writes further achieve performance gains and enable more scalable systems [Gol94]. However even replication of writes might become a bottleneck on servers.
2. Background

Sharding databases [Pri08, Oba09] in such cases, simply means to build partitions, offers another level of scalability but requires an additional software layer.

Small clusters’ hardware reliability always remains a huge obstacle for scalability. Whenever a slaves in such a cluster dies throughput is essentially affected [CAA08].

ODDSE contributes relational fully queryable views on the rather primitive database storage provided by Google that compromises relational primitives. These compromises allow inexpensive and fairly easy scalability by disallowing expensive complex queries and by avoiding strong consistency guarantees.

2.1.2. Distributed storage system - Bigtable

Distributed storage systems such as Bigtable [CDG+06] or HBase [HBa] compromise relational primitives in order to provide scalability. Both systems are similarly designed following design notes given in [CDG+06].

In the following Google’s Bigtable is presented as it was used in this project.

Bigtable leverages the distributed Google file system (GFS) [GGL03], which is designed to scale to thousands of machines. GFS shares the goals of earlier distributed file systems, i.e. scalability, reliability, and availability. Instead of utilizing high-end hardware the system runs on cheap commodity hardware providing fault tolerance. Files are expected to be rather large. The system optimizes access for data-intensive applications [GGL03]. Bigtable on top of GFS offers a simple dynamic model to store structured data but not supporting the full relational data model.

A Bigtable is a distributed, persistent multidimensional sorted data map. The map is indexed by a row key. Data, treated as uninterpreted strings, is maintained in lexicographic order by row key and dynamically partitioned into row ranges. This database schema is radically different from relational schemas. Bigtable doesn’t support selections based on arbitrary columns. Access is limited to gets or scans over the row key, which is the only index that exists.

The key range based partitions are called a tablet and are about 100 MB to 200 MB in size. Being automatically split when growing and redundantly stored on GFS, tablets are the unit of distribution and load balancing [CDG+06]. Especially querying short ranges of data becomes highly efficient as it requires interaction with few machines only.
2.2. Amos II

Amos II (Active Mediator Object System) [RJK04] is a extensible database system that uses a functional data model in combination with the functional relationally complete query language AmosQL.

Queries in the declarative query language AmosQL are specified in SELECT-FROM-WHERE statements and require optimization before execution. The Amos II query engine translates queries first into an object calculus called ObjectLog [LR92]. After several rewrites this representation is optimized by a cost-based optimizer producing an execution plan represented in object algebra. Besides AmosQL Amos II supports SQL through a specific SQL interface. Queries expressed in SQL are first translated to functional AmosQL queries [Jäg05] and then processed as normal.

The data model of Amos II is based on functions, types and objects to represent data. Objects are either of literal types, e.g. charstring, integer and real, collections of objects such as bag of vectors, or of user surrogate types modeling real-world objects. Surrogate objects of user types are given object identifiers maintained by the system. Tables and relationships between different types are modeled by stored functions. Mapped types are derived types in Amos II defined as queries.

Besides stored functions the concept of functions in Amos II provides derived functions to represent queries and views, and foreign functions implemented in an external programming language using the external interfaces provided by Amos II [ER00, Ris00]. In the following usage of foreign functions is limited to the Amos II Java interface. When defining a foreign function Java code is dynamically loaded. The Java Virtual Machine is interfaced with the Amos II kernel through the Java Native Interface to C [RJK04].

Functions in Amos II can be further defined as multi-directional [RJK04, LR92]. Multi-directional functions represent different implementations for a function to compute each of its inverses. Each such implementation is called a type and binding resolved (TBR)
2. Background

foreign predicate [LR92] that is linked to a specific binding pattern. The concept of binding patterns is further explained in [GMUW02].

Multi-directional functions allow a larger class of executable queries and give transparent access from AmosQL to special data structures such as wrapped external data sources. Besides they extend the Amos II query optimizer for better query optimization by a cost model of execution costs and fanouts that can be given for each TBR. The fanout hereby estimates the number of tuples returned per input tuple.

Listing 2.1: A multi-directional function in AmosQL [RFH+05]

```sql
create function sqroots(Number x) as multi-directional
  ("bf" foreign 'sqrts' cost {2,2})
  ("fb" foreign 'square' cost {1.2,1});

/* Usage of sqroots and its inverses */

sqroots(4.0);
  >> -2.0
  >> 2.0

select x from Number x where sqroots(x)=4.0;
  >> 16
  sqroots(4.0)=2.0;
  >> True
```

Listing 2.1 presents a multi-directional implementation of square root and should help to understand the notion of binding patterns. A ‘b’ indicates that an implementation requires a bound variable at the corresponding position in the function’s signature. An ‘f’ on the other hand indicates that a variable can be unbound, meaning free.

The system doesn’t require the implementation of every possible TBR predicate for all ‘b’ and ‘f’ combinations by using a completion algorithm described in [LR92]. This algorithm infers TBR predicates which are covered by other elements. The above example, for instance, doesn’t implement a TBR predicate for the binding pattern ’bb’. Either of the two implemented TBR predicates already cover this pattern. For efficiency the Amos II query optimizer will chose the cheapest available implementation which is the TBR predicate of ‘fb’.

The usage of sqroots and its inverses is rather intuitive as presented in Listing 2.1. Simple foreign functions would only allow the first type of invocation presented.
A core cluster function [Geb99] is a multi-directional function that defines a mapped type. It can also define a view that can be queried with SQL if its named is marked with a ‘#’. Core cluster functions enable Amos II to translate and rewrite SQL or AmosQL queries in order to access wrapped back-end data sources. Mapped types represent an object oriented view on the state of an external data source [Geb99] and can be regarded as foreign types defined by the core cluster function.

OSSE leverages Amos II to compensate for limited GQL query facilities. Multi-directional functions give the opportunity to provide queryable views on Bigtable relations. Mapped types integrate Bigtable relations as types in Amos II and make them queryable in AmosQL. The automatic translation of SQL queries into AmosQL furthermore enables queryable SQL views.

### 2.3. Cloud Computing and App Engine

Cloud computing is a rather new term having become enormously popular during the last two years as illustrated in Figure 1.1. Currently there is no widely accepted definition what cloud computing is all about. Trying to narrow it down [Gee09] gives a round-up of recent attempts of definitions.

Cloud computing in terms of this thesis corresponds to the definition given in [WVLKT08]:

*A computing cloud is a set of network enabled services, providing scalable, QoS guaranteed, normally personalized, inexpensive computing infrastructure on demand, which could be accessed in a simple and pervasive way* [WVLKT08].

Inexpensive computing infrastructure thereby refers to pay-as-you-go resources as a service eliminating high initial investments and generally reducing continuous costs for IT infrastructure. Concentrating computational power in huge data centers allows the provision of such inexpensive services as infrastructure becomes economical to administer and workload peaks are easier to balance among customers. Computing clouds are furthermore based on cheap commodity hardware and virtualization technology making it easy to share resources [Wei07]. Distributed fault-tolerant file systems such as GFS provide the required reliable and scalable storage facilities on top [WVLKT08].

Virtualization technology allows to add resources from a vast resource pool to applications nearly from one moment to another, providing a highly adaptive infrastructure on demand.
2. Background

Figure 2.2.: A classification of cloud services. Higher level services are generally build up on lower level cloud services.

[WVLKT08]. Resource provisioning is thereby automated based on continuous monitoring to ensure scalability and Quality of Service (QoS) guarantees. Amazon’s Elastic Compute Cloud (EC2) subtly instills this adaptivity in its name.

User centric interfaces, often related to Web 2.0 services and applications, allow easy usage. Setting up an App Engine on Google, for instance, takes less than a minute immediately having the powerful cloud service available.

As illustrated in Figure 2.2 cloud services are generally classified in three levels which typically build up on each other. Amazon’s EC2 is a well known example for Infrastructure as a Service (IaaS). The Google App Engine, investigated in this project, is an example of a Platform as a Service (PaaS) ensuring scalable Python web applications [Law08]. Applications and service on the top level are often referred to as Web 2.0.

[AFG⁺09] discusses some obstacles of cloud computing. Relying on a single cloud provider raises the question of Availability of Service. [AFG⁺09] suggests utilizing a virtual infrastructure from various providers to avoid single points of failures. Another difficulty is the issue of Data-lock in as addressed of this project regarding Google App Engine. Last but not least Data Confidentiality and Auditability becomes a major concern when moving sensitive data into the cloud and should be taken into account when using ODDSE to move huge data volumes to Bigtable.
2.3. Cloud Computing and App Engine

2.3.1. Google App Engine

With the App Engine [Goo08] Google offers a Platform as a Service (PaaS) allowing web developers to build highly scalable and reliable web applications. Running on virtual machines in Google’s cloud, such applications benefit from automatic scaling and load-balancing. The usage of Google App Engine is for free within certain quotas. Additional quota when exceeding available free resources has to be purchases from Google.

Currently developers can choose among two application environments, the initial Python framework and a Java sandbox recently added in April 2009. The Java environment is not further discussed in this report.

Most interesting, developers are provided with a highly scalable cloud storage system called the App Engine Datastore, built on Google’s distributed storage system Bigtable [Bar08].

For ease of development Google offers a SDK allowing local development and testing. The SDK includes a tool for bulk uploads of data, which is evaluated in this thesis with respect to performance and compared to ODDSE.

Concerns of lock-in have been addressed regarding Google’s App Engine as earlier mentioned [O’G08, O’R08, Dis08]. Recently, however, there had been successful attempts to set up a modified App Engine on top of EC2 by adapting the open source App Engine API. Underlying Google technology was successfully replaced by open source software. One such system is App Drop [And08] which uses MySql as storage system instead of Bigtable. Independent from the infrastructure lock-in this thesis presents another solution to solve the data lock-in problem.

The App Engine Datastore

The App Engine Datastore provides a high level API giving an abstract view on Bigtable and hence offers persistent scalable storage of data as well as limited query facilities by means of Google’s query language GQL.

The Datastore data model is entirely managed in Python and differs from the relational data model, rather showing similarities with the earlier described ER data model.

---

1MySql is a relational DBMS and available from [http://www.mysql.com/](http://www.mysql.com/)
Based on an API Python classes are created reflecting entity types having certain properties similar as the ER model. Instances of entity types are stored in Bigtable. The above Listing 3.14 presents the definition of an entity type \textit{Person} in Python.

Recall that Bigtable gets and scans are possible only on the single existing index, the row key. Therefore, to allow queries on any single entity property, indexes are created on every attribute. These indexes are separate Bigtable tables mapping a property value to corresponding entity keys. In such an index, each row contains the index data and an entity key.

Entities are stored as serialized strings in the single column of a special generic schemaless entity table for all entity types [Bar08].

Using GQL Bigtable entities can be queried. However, GQL is not a relationally complete language and the queries are obviously limited to existing indexes.

Limitations of Bigtable and further restrictions imposed by the Datastore API are important for understanding ODDSE and these limitations are thus explained thoroughly in the remainder of this section.

- The result set of GQL queries is limited to 1.000 entities.
- The Datastore doesn’t support arbitrary queries for above reasons. Filter queries must be applied on single physical indexes to select rows based filtered properties in order to enable an efficient scan on a minimal number of tablets (distributed row ranges). The number of additional indexes on composite properties is critical regarding space consumption and is moreover generally limited by the system.
- GQL is null-intolerant meaning empty properties are always evaluated to false. This is due to the fact that entities with an empty properties are not included in the corresponding index.
- Inequality filters are allowed on one single attribute in a query.
• The attribute used in an inequality filter must be the first attribute of the query’s sort order if specified.

• GQL has the following functional limitations (see Appendix A for the GQL reference):
  
  – GQL doesn’t provide means for INSERT, UPDATE and DELETE. These operations thus require explicit looping in Python calling the API functions put() or delete() on every entity object.

  – There’s no aggregation function support.

  – GQL does not support string filter statements such as LIKE.

  – GQL has no native JOIN facilities and thus allows queries over single entity types only.

In ODDSE multi-directional functions in Amos II are used to implement fully views on each single entity type stored in Bigtable. The access pathes to Bigtable for such a multi-directional function are implemented respecting the above functional limitations and avoiding the risk of expressing unsupported GQL queries.

In addition to the above functional limitations quotas are imposed on resource usage by the App Engine. These quotas are reset every 24 hours. Per-minute quotas enforce a more uniform consumption and disallow high usage peaks.

• The runtime of every script is limited by timeouts. The timeout interval is thereby dynamically reduced for a script with high CPU usage.

• CPU time is generally limited. The number of requests with high CPU usage is furthermore restricted.

• Calls to the Datastore API are limited by even more restrictive timeouts as well as quotas in terms the data volume transferred.

• The amount of stored data is limited consequently restricting the number of available indexes.

• Outgoing and incoming bandwidth is limited.

ODDSE is capable of dealing with these resource limitations. In order to avoid long running requests with high CPU usage and therewith the high risk of running into timeouts
2. Background

or violating quotas, queries are optimized during runtime and dynamically narrowed down into smaller subqueries. To reduce the risk of Datastore timeouts, such subqueries might be further split into smaller subqueries when calling the Datastore API. For space efficiency ODDSE moreover uses only the default indexes on single columns rather than creating additional indexes on composite columns.

ODDSE encapsulates entity types stored in Bigtable providing a transparent view that can be queried in SQL. In combination with SQL the relational terminology as earlier described is therefore used throughout the rest of the thesis. App Engine entity types are hence referred as *Bigtable relations* having certain *attributes*. Bigtable entities are called *rows*.
3. ODDSE (Optimized Distributed DataStore Exchange)

ODDSE aims to provide a transparent way to access the distributed storage system of the Google App Engine. Being aware of the App Engine’s as well as Bigtable’s limitations as earlier described in Section 2.3.1 ODDSE provides a reliable and failure safe query interface to exchange data with the App Engine Datastore.

ODDSE is implemented as a wrapper for Amos II. It gives an integrated view on the Datastore that allows querying data in the Datastore with full SQL (or AmosQL) functionality as well as combining the external data with other Amos II data.

This section gives a brief overview of ODDSE and its architecture to give a closer understanding of query processing in ODDSE.

3.1. System overview

Figure 3.1 illustrates the overall architecture of ODDSE. Being a wrapper for Amos II, ODDSE provides functionality to query the distributed storage system Bigtable. Therewith an integrated view on Bigtable is offered that allows general query facilities in terms of SQL as well as queries in the Amos II specific, functional query language AmosQL. The provided query facilities through ODDSE exceed the native query facilities of GQL by doing none GQL-supported query evaluations using the Amos II query engine running in the client.

The queries specified in SQL or AmosQL are entered to the Amos II user interface as illustrated in Figure 3.2. The following listing 3.1 gives a simple example of such an AmosQL query. During processing by the Amos II query engine an execution plan is produced where calls to ODDSE are implemented by calling the OSSDE wrapper as a
3. ODDSE (Optimized Distributed DataStore Exchange)

Figure 3.1.: Architecture overview of ODDSE. The system provides an SQL, an AmosQL, and a bulk upload interface to access Google Bigtable by means of an ODDSE server on top of Google App Engine.

Listing 3.1: Querying data in AmosQL

```sql
select ssn(p) from Person p where name(p)=’Moritz’;
```

SQL queries are translated by the SQL preprocessor of Amos II into AmosQL queries that are processed as other AmosQL queries. Listing 3.2 shows the equivalent of the previously defined AmosQL query using the SQL interface. It is translated to an AmosQL query analogous to the one in Listing 3.1.

Listing 3.2: Querying data using SQL

```sql
SQL(
    "select ssn from Person where name=’Moritz’"
);
```

In order to wrap GQL-based data sources, ODDSE provides a Schema Manager as illustrated in Figure 3.3. This component is capable of transforming Bigtable data schemas into a corresponding Amos II ODDSE schema representation and vice versa. The ODDSE schema includes for instance an ODDSE data dictionary and an AmosQL function providing access to Bigtable by means of the ODDSE wrapper. In Section 3.3.1 the various types of metadata are thoroughly explained.
3.1. System overview

Figure 3.2.: Query processing in ODDSE: The ODDSE client splits queries into chunk queries in order to query Bigtable by means of an ODDSE server in parallel using simultaneous connections.

The schema manager provides a transparent view of the wrapped external data source to the user by hiding from users the complexity of schema exchange and transformations by automatically creating the ODDSE schema. This means that ODDSE users will usually not be aware of that they are querying an external Bigtable relation, as it is queried transparently the same way as a local table.

ODDSE’s schema tools can be called from the Amos II user interface as foreign functions. The allow, e.g., creation of Bigtable relations that are instantly wrapped. Such system functions exchange metadata between ODDSE clients and servers.

Listing 3.3: Creation of a new Bigtable relation through ODDSE

```java
/* creation of a new Bigtable relation in ODDSE */
createBigtable(
    "Person",  // relation name
    {"ssn","name"},  // attributes
    {"Charstring","Charstring"},  // attribute types
    {"ssn"}  // key attributes
);  
```

Listing 3.3 shows the creation of a new Bigtable relation Person in the App Engine by
ODDSE. The relation *Person* is instantly wrapped and accessible from ODDSE.

Last but not least ODDSE offers not only query facilities, but also bulk upload of data into a wrapped Bigtable relation. The bulk upload is illustrated in Figure 3.4. In combination with simple select queries to download entire relations the presented system is therewith capable of performing backup and restore operations on Bigtable relations. Data can be uploaded from files with character separated values (CSV-files). The separation character of such files can be set in the ODDSE configuration (see Appendix C.1).

Listing 3.4: Data file for bulk upload

```
450614−338|Max Mustermann
620120−735|William Nilsson
750624−523|Karl Maier
821110−621|Michael Smith
```

Listing 3.5: Data upload in ODDSE

```
/* Bulk upload of person.csv into relation Person in ODDSE */
bigtableUpload(
    "person.csv", // bulk data file
    "Person", // relation name
    {"ssn","name"} // attribute mapping
);
```

Listing 3.5 shows the bulk upload of a data file into the relation *Person*. A bulk data file
3.1. System overview

Figure 3.4.: Bulk uploads in ODDSE: The ODDSE client splits input data into data chunks and uploads these to Bigtable by means of an ODDSE server using several simultaneous connections.

could look like the one presented in Listing 3.4. Attributes without a mapping are set to null in Bigtable. The mapping of all key attributes is of course required.

The previously presented examples hopefully helped to enlighten the ease of use provided by ODDSE when accessing Bigtable relations in the App Engine Datastore. The interested reader will find more examples and details about supported use cases in Section 3.2.

Under the covers of the query interfaces ODDSE heavily optimizes the communication with Google App Engine. Figure 3.2 illustrates the architecture of ODDSE connotating the usage of several simultaneous server connections in order to improve data throughput.

As shown previously in Figure 2.2 Google App Engine offers a Platform as a Service (PaaS). The ODDSE server is build on top of this platform and offers fault-tolerant interaction with the App Engine Datastore in Bigtable. Utilizing the underlying cloud services the ODDSE server is able to offer scalable services. In order to achieve scalability these services work on parallelizing access to rather small chunks of data. The services are:

- Schema interchange between Amos II (ODDSE) and Bigtable.
- Bulk upload of data for storage in Bigtable.
- GQL query processing in Bigtable, in contrast to the full query facilities in the ODDSE client.
Bulk uploading and query processing services are orchestrated by the ODDSE client, which in parallel works on small partitions of the original request. These partitions are called *chunks*. The partial results of chunks from parallel service invocations are then combined asynchronously to present the entire result to the user. This allows processing of large data volumes. Figure 3.2 illustrates the way the ODDSE client uses the above services to access Bigtable.

Due to various restrictions and limitations, processing on the App Engine is rather unreliable as it might be interrupted by App Engine failures. Query processing (including bulk uploading) on the ODDSE server therefore provides fault-tolerance by returning information to the client on how to resume a failed query in case of failure. This information is utilized by the *resumable query manager* of the ODDSE client in order to continue query processing or bulk uploading from the point of failure.

Amos II includes a full database management system itself, which is used by ODDSE for management of its metadata as illustrated in Figure 3.2. Part of the metadata are Bigtable metadata described by *core cluster functions* that define views of accessed Bigtable relations. Based on the core cluster functions various rewrites are made for scalable query processing. The rewrites are done by the Amos II query processor before sending a GQL query request to ODDSE using the external Java interface of Amos II [Ris00, ER00].

Processing in the ODDSE client is heavily dependent on the various types of maintained metadata. Metadata describing a wrapped Bigtable relation is stored both on the Amos II client and on the App Engine server. The client’s data dictionary is managed by Amos II. It also includes location information to handle data from several App Engine servers at different locations. On the server side the data dictionary is stored in Bigtable itself. To improve performance some caching mechanisms are implemented in the App Engine server in order to speed up access to the data dictionary.

Tracking status information of an App Engine server helped to improve the overall performance of the system. This status information is managed by a *runtime optimizer* and relies on statistics about the App Engine based on the following two assumptions:

- A large number of timeouts of App Engine service invocations indicates a high workload on the server side.
- A large number of quota failures indicates the oncoming violation of available quotas on your App Engine server.
Figure 3.5.: Chunk queries, as presented in this figure, are queries that return a partition of a given owner query. The combined result of all chunks is equivalent to the result of the owner query. ODDSE uses chunk queries to process operations in parallel on the App Engine.

Every time an App Engine service invocation fails the runtime optimizer updates its App Engine server status according to the above assumptions. Based on the tracked status, the runtime optimizer adapts a set of system parameters. In particular these are the number of simultaneous server connections used by the query scheduler and the target chunk size for the query partitioner.

Using statistical data on wrapped data sources allows furthermore efficient preprocessing of queries. Such statistical data includes the number of rows of a queried relation and data histograms on single attributes. The query partitioner analyzes queries and applies statistical data, if available, in order to build small chunk queries of a target size given by the runtime optimizer. These chunks enable massive parallelization to scale up the ODDSE services running on the App Engine cloud platform. The data returned can be sent to the statistics manager component to create data statistics for further operations. This is currently only done when specifically analyzing a Bigtable relation.

3.2. Supported Scenarios

3.2.1. Schema integration and exchange

Figure 3.3 presents the schema manager that wraps GQL-based data sources for usage in ODDSE. The component is capable of transforming Bigtable data schemas into an ODDSE specific schema representation and vice versa. Hiding the complexity of schema transformations from users, the schema manager provides a transparent view on wrapped Bigtable relations. For that reason users usually will not be aware of the fact that they are querying an external Bigtable relation.
3. ODDSE (Optimized Distributed DataStore Exchange)

As earlier described in 3.1 and illustrated in Figure 3.2 metadata is stored in both the Amos II client and the App Engine server. Wrapping of Bigtable relations in Amos II is achieved through target binding patterns described by core cluster functions which define views of external data sources in Amos II that can be queried with SQL or AmosQL. All necessary binding patterns are generated automatically by the schema manager.

The examples below are intended to give an overview on how to wrap or create external Bigtable data sources in ODDSE in order to query them with Amos II.

Listing 3.6: Creation of a new Bigtable relation (Listing 3.3)

```java
/* Creation of a new Bigtable relation in ODDSE */
createBigtable(
    "Person", // relation name
    {"ssn","name"}, // attributes
    {"Charstring","Charstring"}, // attribute types
    {"ssn"} // key attributes
);
```

In the above example the core cluster function #Person is automatically created by the schema manager. The metadata of the new relation is furthermore published to the App Engine server at the given location in order to create it in Bigtable as well as to store it in the server’s data dictionary. The multi-directional core cluster function #Person provides all necessary binding patterns to query the newly wrapped relation using the SQL view Person(ssn, name) The # indicates to Amos II that Person is an SQL table.

To enable object-oriented AmosQL queries, additionally mapped types are generated named Person and Customer. They integrate Bigtable relations for usage in AmosQL and are elaborated closer in Section 3.3.1.

Listing 3.7: Wrapping all relations from an ODDSE server (AmosQL) label

```java
/* Access and wrap all relations from the specified App Engine */
accessBigtable(
);
```

The above example ?? shows how existing Bigtable relations on an App Engine are accessed and wrapped for usage in ODDSE. For each relation present in Bigtable on the specified App Engine a core cluster function, SQL view, and a mapped type are generated.
3.2. Supported Scenarios

Listing 3.8: Copy a Bigtable relation schema to a new location (AmosQL)

```plaintext
/* Copy an existing wrapped Bigtable relation schema to a new location */
createBigtable(
    "Person", // relation name
);
```

Last but not least the schema of an already wrapped Bigtable relation from a specific App Engine can be copied to another location. The relation’s metadata is copied to the App Engine at the specified new location. Based on the copied metadata the relation is then created.

### 3.2.2. Query processing

A language reference for GQL is provided in Appendix A. This query language does not provide more sophisticated queries such as, for example, periodical business analyzes. However, running a business on a remote cloud infrastructure makes such analyzes essential in order to evaluate performance. The absence of business critical data for decision support often imposes wrong decisions being made.

Business analyzes are furthermore often also dependent on the integration of data from various sources. Sensitive critical customer data for instance such as credit card numbers might not be intended for storage in a more or less uncontrollable cloud environment [BYV+08]. On the other hand such critical data might be required when performing certain queries.

To alleviate these GQL limitations, the transparent view on Bigtable data sources provided by ODDSE supports complex queries expressed in common SQL even combining data from heterogeneous data sources available through Amos II.

On top of the expressiveness of queries some important functional requirements have to be ensured. The App Engine is rather unreliable when dealing with huge amounts of data. Enabling extensive query facilities requires basic reliability and fault-tolerance. ODDSE is able to guarantee these functional requirements for wrapped Bigtable sources.

In order to achieve high performance, optimization of the interaction with the ODDSE services in Google’s cloud becomes crucial. Especially the likelihood of expensive App
Engine failures on costly operations decreases the system’s throughput enormously and requires careful design. The system also needs to deal with changing workload on the server as well as declining quotas while operations proceed. Therefore a dynamic fault-tolerant interaction management is required. The ODDSE wrapper (Figure 3.2 and 3.4) contains such interaction management based on the resumable query manager and the runtime optimizer.

In the following a selection of queries is presented in order to illustrate the functioning of ODDSE. All these queries are expressed in SQL and can always be replaced by an equivalent AmosQL query.

**Listing 3.9: Selection of an entire Bigtable relation**

```sql
/* Selecting all rows from Bigtable relation Person */
SQL(
    "select * from Person"
);
```

The above Listing 3.9 presents the most general binding pattern for wrapped data source. Selection of all rows of a Bigtable relation corresponds to a download of the relation from the cloud storage.

**Listing 3.10: Simple SQL select**

```sql
SQL(
    "select * from Person where first_name in
     ("'Daniel', 'Danielle', 'Dani', 'Daniela')
     and age > 30 and age < 40
     order by first_name"
);
```

Listing 3.10 shows a rather simple SQL query that looks up a person given some knowledge about the person’s name and age. This query exceeds the capabilities of GQL, but can be efficiently processed using ODDSE.

**Listing 3.11: SQL join**

```sql
SQL(
    "select first_name, last_name, cc_no, sum(price)
    from Person, Order, CreditCard
    where Order.ssn == CreditCard.ssn"
);  
```

26
3.2. Supported Scenarios

Listing 3.11 gives a more complex example challenging the capabilities of the Bigtable wrapper and Amos II’s mediator abilities. This example is certainly not made up out of thin air. Running a small online shop on Google App Engine for instance such a query could be rather useful. For various reasons the example is interesting:

First of all let’s consider the two Bigtable relations Person and Order to be the wrapped Bigtable sources. By contrast, data stored in the relation CreditCard felt too sensitive to be stored remotely. Hence this data is available through a locally wrapped relational database.

Having this in mind one can see how transparent data integration works using ODDSE with the mediator Amos II. Heterogeneous data sources are easily combined using a common SQL join operation. Users moreover don’t even have to be aware of any differentiation between wrapped and internal data sources.

After joining the retrieved data from three data sources the intermediate result is grouped by each person in order to apply an aggregation function on the purchased amounts of all orders of the last day.

As one can see Join operations as well as Group BY and aggregate functions are well supported by the provided SQL interface giving huge query power and usability.

3.2.3. Bulk loading of data

Taking into account as close as possible the App Engine’s quota restrictions and using a mechanism of recover and resume, the runtime optimizer enables efficient bulk uploading of huge datasets into Bigtable. The performed tests proved bulk loading hereby to be highly critical with regard to optimization. Expensive writes and the way the App Engine performs INSERTs require careful attention and deeper insights into the Datastore API. Documented and suggested ways by [Goo08] proved to be rather ineffective.

Bulk loading was developed first to provide a sufficient set of data to evaluate ODDSE’s runtime optimizer performance. For the tests, data dump files generated by the TPC-
H database benchmark suite [Cou99] were used. These dump files consist of character
separated values, the CSV format.

CSV files are simple to create making ODDSE’s bulk loader suitable for restoring data
backups or performing initial uploads of data. There are also a couple of other useful
applications described in Chapter 4.

ODDSE currently supports two approaches of bulk loading. From a user perspective both
approaches seem to be rather similar. Nevertheless the following two examples illustrate
the usage of both. Their difference will be explained and evaluated later in Chapter 4.

Listing 3.12: Bulk uploading of data (AmosQL)

```c
bigtableUpload(
    "/TPC-H/dbgen/customer.tbl",
    "tpchCustomer",
    { "c_custkey", "c_name", "c_address", // attribute mapping
       "c_nationkey", "c_phone", "c_acctbal",
       "c_mktsegment", "c_comment", "" }    // ignore last
);
```

The above listing shows the simple bulk loading interface. The bulk upload requires an
attribute mapping of columns in the bulk file to attributes. Columns mapped to an empty
string are hereby ignored. Unmapped attributes are set to null in Bigtable. Key attributes
cannot be null and have to be mapped.

The operation is optimized as briefly described earlier. If, however, the data dump file for
upload is key sorted, the following advanced bulk loading operation can give additional
performance:

Listing 3.13: Randomized bulk uploading of data (AmosQL)

```c
bigtableRandomizedImport(
    "TPC-H/dbgen/customer.tbl",
    "tpchCustomer",
    { "c_custkey", "c_name", "c_address", // attribute mapping
       "c_nationkey", "c_phone", "c_acctbal",
       "c_mktsegment", "c_comment", "" }    // ignore last
);
```

28
This operation mainly differs from the previous one in the order data chunks are uploaded. The randomized bulk uploading approach will partition the whole set of data first in order to randomize the data chunks. Due to the way load balancing is achieved among Bigtable’s storage nodes the randomization helps to achieve better performance when inserting rows into Bigtable. The specific reasons are discussed later in Section 4.1.3.

The performance benefit gained from randomization is not for free however as it requires significantly more memory on the client side to store handles for each data chunk prepared in advance. Such a data chunk handle is a tuple (file offset, chunk size). The data itself is however not stored in main memory. Nevertheless, `bigtableRandomizedUpload` should be only used when allocating additional memory to the Java Virtual Machine on the ODDSE startup.

### 3.3. Architecture

#### 3.3.1. Metaschema

In order to understand the internal functioning of ODDSE and in particular ODDSE’s client, some knowledge about the metadata maintained in the system is crucial. The following list gives a detailed description of the different types of metadata utilized and maintained by the system:

![Metadata Diagram](image)

Figure 3.6.: The type of metadata maintained in ODDSE clients and servers. It stands out that the server only maintains a data dictionary while the client utilizes a variety of different metadata.
Figure 3.7.: The different types of metadata created and adapted by the ODDSE client, and how they are utilized by different components during query processing and bulk uploading. The statistics manager is only present for query processing.

**Data Dictionary** Both the ODDSE client and server store a data dictionary. As expected, this data dictionary contains type information including relations along with their attributes and key attributes. While the client needs to store location information this information is unavailable on the server.

Due to the fact that the App Engine currently doesn’t allow server initiated actions (cron jobs), location information on the server can’t be utilized anyhow.\(^1\)

On the client side the data dictionary is used to enable query processing as well as bulk uploading and to direct a query request to the appropriate ODDSE server running in the App Engine platform. Regarding query processing, key awareness showed to be of great importance. The performance impact achieved choosing key access paths, where possible, corresponds to a cost model provided by *multi-directional functions* as described in Section 3.3.1.

\(^1\)However the provision of cron jobs is stated on Google’s Road Map and shall be provided in the near future (March ’09))
3.3. Architecture

On the server side data dictionary entries, each called a *DbRelation*, provide schema flexibility. These entities are stored in Bigtable as illustrated in Figure 3.2.

The App Engine Datastore API requires a static schema that depends on Python classes inheriting from an abstract model class provided by the App Engine Modeling API as shown in Listing 3.14 below. Every such class represents a relation in Bigtable and is ready for storage and querying as soon as being defined.

Listing 3.14: A Datastore model class in Python

```python
# Definition of the App Engine type Person that corresponds to the # relation Person(first_name, last_name, age) wrapped in ODDSE
class Person(db.Model):
    first_name = db.StringProperty(required=true)
    last_name = db.StringProperty(required=true)
    age = db.IntegerProperty()
```

This restricts creation of new relations to developers having administrational access to an ODDSE server: With every schema change a piece of Python code has to be uploaded or modified.

*DbRelation* instances are dynamically translated into Python classes maintaining the relations. In contrast to model classes *DbRelations* can be created from any ODDSE client using *createBigtable* without uploading a single piece of Python code. This gives great flexibility and highly increases usability. On top of that a *DbRelation* performs the appropriate type mapping between Bigtable and Amos II and backwards. For performance improvement the dynamic classes are cached in *Memcache*, a highly available caching system provided on the App Engine platform. Caching of dynamic classes avoids too frequent translation of *DbRelations* but might delay schema propagation of changes in some cases, however.

The ODDSE data model supports the data model of Amos II, with some minor restrictions.

Composite keys are supported by ODDSE. This common database feature is not available on Bigtable, but important in order to support transparent integration of Bigtable relations in Amos II. *DbRelation* classes are capable of dealing with such keys, though. If a composite key is defined for a type a hidden system property is automatically added. This property is used to store the concatenated key value. If
3. ODDSE (Optimized Distributed DataStore Exchange)

there are non string key members the resulting key value might cause a unexpected but still consistent sort order.

**Core cluster functions** For each accessed Bigtable relation the ODDSE *schema manager* generates automatically a core cluster function with different GQL query expressions for each different binding patterns. This is illustrated in Figure 3.7. The query strings, implementing TBR predicates (Section 2.2), define the access paths supported by the Bigtable relation in terms of existing indexes. They include a general access path for queries having no variable bound (i.e. to scan the Bigtable relation), a pattern binding supporting the (composite) key of the relation, as well as all binding patterns having a single attributes bound. Utilizing the earlier presented completion algorithm for multi-directional functions it is not required to implement every possible TBR predicate. An example of such a automatically generated core cluster function is presented in Listing 3.15.

The first two TBRs are supported by Bigtable. TBRs for single attributes are possible as the Datastore API automatically maintains indexes in Bigtable on every single attribute of a Bigtable relation. All other TBR predicates however are usually incompatible with Bigtable and will result in an *Index Error* returned whenever a query is not supported by an existing index. The generation of the different TBR query expressions along with the completion algorithm will guarantee that the query never fails, since all binding patterns are guaranteed to be supported by GQL and Bigtable.

Listing 3.15: The core cluster function #Person

```haskell
/* Core cluster function for Person */
create function #Person() ->
Bag of <Charstring ssn key, Charstring name> as multi-directional
("ff" select ssn, name where {ssn, name} =
  queryBigtable("select * from Person";{},))
("bf" select name where {ssn, name} =
  queryBigtable("select * from Person where ssn=%s",{ssn}))
("fb" select ssn where {ssn, name} =
  queryBigtable("select * from Person where name=%s",{name}));
```

Along with the core cluster function the schema manager creates a mapped type when integration a new data source. This foreign Amos II type, defined by the core
cluster function, provides full AmosQL query facilities.

Listing 3.16: The mapped type *Person*

```java
create_mapped_type("Person",{"ssn"},{"ssn","name"},"#Person");
```

Listing 3.16 shows an example of a definition of the mapped type called *Person*, which is automatically generated by ODDSE. As noticeable from the listing the last parameter passed to the Amos II function `create_mapped_type` is the core cluster function `#Person`, which was defined in Listing 3.15. Using the multi-directional bindings provided by `#Person` Amos II is able to resolve the appropriate TBR predicate depending on free and bound variables to query the APP Engine whenever the mapped type *Person* is used.

Listing 3.17: Querying data using mapped types in AmosQL

```sql
select ssn(p) from Person p where name(p)=
'Moritz';
```

Listing 3.17 illustrates the usage of mapped types for query purpose in AmosQL. Having *ssn* unbound and *name* bound to 'Moritz' the binding pattern for this query is obviously 'fb' leading to the last TBR predicate `#Person^b` to be executed.

**Foreign functions** ODDSE extends Amos II with several *foreign functions* to provide access to ODDSE’s functionality. Foreign functions are for instance used by the core cluster functions to access the ODDSE wrapper by sending GQL strings to the App Engine Datastore for evaluation. For example, in Listing 3.15 the foreign Java function `queryBigtable` of the ODDSE wrapper is called. Most of the foreign functions are used to manage the metadata itself. Appendix C gives an overview of all (foreign) functions provided by ODDSE.

**ODDSE parameters** System parameters maintained ODDSE can be classified into rather static system parameters defining a general system configuration as well as highly adaptive parameters describing the client-server interaction. The latter parameters, including the size of data and query chunks as well as the number of App Engine connections, are maintained by the runtime optimizer. They are frequently updated while executing queries to the back-end App Engine. The App Engine is exposed to highly dynamic workloads in addition to restricted and continuously changing quotas, to which ODDSE must adapt.
Adaption of dynamic parameters is shown in Figure 3.7 and described in detail when presenting the ODDSE runtime optimizer in Section 3.3.3. The adoptions are generally done based on a query’s type such as general queries or bulk uploads.

ODDSE parameters are accessible from the Amos II query interface. Appropriate functions to adapt parameters are described in Appendix C. Appendix C.1 provides an overview of available system parameters and their impact on the system. This table should be considered before changing any working parameter set.

**Statistics** The query partitioner is dependent on certain statistics of relation properties in order to split queries returning large result sets into queries returning smaller chunks that can be executed simultaneously. Such chunking helps to decrease latency on the one hand. On the other hand it immensely decreases the risk of expensive App Engine failures due to long exhaustive operations running into timeouts.

![Equi-width histogram](image1)

![Equi-depth histogram](image2)

**Figure 3.8:** This example compares equi-width and equi-depth histograms. As illustrated on the left buckets of equi-width histograms are based on equally sized value intervals (here of size one). Equi-depth histograms, by contrast, provide buckets of similar depth in terms of included rows. The equi-depth buckets presented on the right side correspondingly all contain 10 rows.

The statistical data is maintained by the statistics manager as illustrated in Figure 3.7. Based on these statistics the query partitioner provides chunk generation either assuming a flat distribution of values of a column between known interval end points, or in correspondence to more sophisticated histograms maintaining the distribution of values per column. These histograms allow selectivity estimation of value ranges of high quality and hence support range queries very well.

Figure 3.8 illustrates the difference between equi-width and equi-depth histograms. ODDSE’s wavelet based histograms are based on equi-width histograms of very fine granularity which are transformed into a compressed wavelet domain. Based on the multi-resolution properties of the created wavelets equi-depth histograms of almost
any chunk size, respectively bucket size, demanded by the runtime optimizer can be created.

Query chunks are build correspondingly to equi-depth buckets. A query chunk thereby is a query that returns a partition of a given owner query.

The following types of statistical data are maintained in the system:

- **Entity statistics** provide the number of entities of a given kind.
- **Property statistics** keep track of upper and lower values of properties of a given kind.
- **Wavelet coefficients** encoding equi-width histograms of relation attributes. The usage of wavelets and their transformation is described in detail in Section 5.1.3.

### 3.3.2. The ODDSE communication protocol

Initial research for an appropriate protocol for the ODDSE client-server communication was done first. [SW00] describes the XML-centric information server Tamino which implements such an XML based protocol. Offering a XML specific query language XQL, Tamino not only provides retrieval from native XML storage, but also integration of ordinary SQL databases. This system served as a first reference for the ODDSE protocol.

Nevertheless following [NJ03, WG08] using XML as data interchange format is strongly discouraged for the present use case. [NJ03] evaluates the performance of IBM’s XML4C SAX parser and relates it to relational database performance. According to the published results SAX parsing of XML raises the costs of a database transaction without any schema validation already by a factor of 2 to 3. Due to this poor performance in combination with the App Engine’s exceptionally limited resources in terms of CPU cycles and restrictive process timeouts, a more lightweight microformat is essential to build a scalable system.

Analyzing the demands of an appropriate protocol there are three important needs to be addressed. First of all the protocol has to be able to transfer any possible query expressible in Google’s query language GQL. The protocol has to be further capable of sending any result, that is any set of tuples, back to the client as well as loading similar data on the server. Finally, any kind of server failure has to be uniquely encoded to be sent to the client for resumption.
3. ODDSE (Optimized Distributed DataStore Exchange)

Figure 3.9: The client-server communication in ODDSE. Requests to an ODDSE server are send as simple HTTP POSTs. The answer is returned as lines of comma separated values. The first line defines the schema of the returned data. Special markers indicate server side failures and allow resumption in the client.

Google’s query language GQL is indeed very simple (see Appendix A). A query basically consists of an entity type, a set of unnested filters, and a certain sorting. Any such query can be mapped to a set of POST parameters as illustrated in Figure 3.9. Using the App Engine’s own web application framework webapp POST parameters can be processed in a convenient way.

Listing 3.18: Bigtable query object creation in Python

```python
# Query object from the Datastore API to access
# the relation Person in Bigtable
query = db.Query(Person)
query.filter('sex =', 'm')
query.filter('ssn >', '99')
query.order('ssn')

# add restrictions for chunk query
query.filter('ssn >=', '990614-435')
```

...
As queries are generally split in up into hundreds and thousands of chunks in order to be more efficient, Python query objects provided by the Datastore API are cached on the ODDSE server in the App Engine Memcache. These query objects, such as the example presented in Listing 3.18, hold a translated prepared query statement and are uses to perform queries against Bigtable. The query objects are tunable by accepting additional chunk interval parameters. Hence all parameters presented in the background of Figure 3.9 need to be transferred to the ODDSE server only for the the first ODDSE server request. Successional chunks, identified by the queryId of their owner query, profit from the query object cached in the ODDSE server. The system attaches to such a query object two filters according to the chunk bounds so that the chunk result can be easily retrieved. The same POST parameter based mechanism applying caching for better performance is used to bulk upload data on the server.

A server response consists of lines of comma separated values of which the first line defines corresponding value types. Special markers moreover indicate App Engine failures or exceptions in detail and hence allow resumption in the client.

**Failure handling**

When being close to the App Engine’s resource and quota limitations for optimal performance it is crucial to deal appropriately with violations and failures on the ODDSE server in order to provide reliability and scalability.

As reasons for failures vary broadly, different types of failures have to be distinguished by the ODDSE client. It is essential to the runtime optimizer to adapt the system to the current status.

The following lineup gives a classification of failure types handled in ODDSE:

**Timeout failure** The maximum runtime of App Engine applications is limited to a few seconds. When exceeding this restriction the server-side processing is interrupted. The ODDSE protocol is able to overcome this suspension by returning a resume point.
3. ODDSE (Optimized Distributed DataStore Exchange)

from where to continue the current query. To avoid further timeouts the runtime optimizer in the ODDSE client tries to decrease the chunk size of succeeding queries.

**Deadline exceeded failure** Long lasting API calls might result in a *deadline exceeded failure*. These failures are regarded as *timeout failure* und handled the same way.

**Over quota failure** All CPU violations of App Engine API calls, each having its own CPU time contingent, or on the App Engine itself are reported as over quota failures. On the server-side this failure type is handled the same way as timeout failures. Client processing requires further considerations. To recover from a quota failure all resource allocation is postponed by the ODDSE client for a short interval as Google restricts resource consumption using per-minute quotas. The adaptive recovery interval is specified by the runtime optimizer and correlates to the frequency of *quota failures*. At the same time parallelization is reduced to decrease the probability to running out of quota soon again.

**403 - forbidden** Exceeding quotas might result in a HTTP status response 403 instead of returning a *over quota failure*. Resumption is handled accordingly.

**Index errors** As earlier described talking about Bigtable’s limitations in Section 2.1.2 queries are always executed on a physical index covering the given query. Whenever one tries to execute a query that is not backed by an existing index the query will fail returning an *index error*.

This however can only be the case when using ODDSE’s internal GQL interface. Using the high level interfaces for SQL and AmosQL provided by Amos II such a failure is not possible, as the ODDSE schema manager only maps supported access paths (binding patterns) to the GQL interface using multi-directional foreign functions.

**General exceptions** Even though the ODDSE Protocol can resume processing from any error, there is no way to always resume from a failed state on the App Engine. To provide transparency in such cases general exceptions are handed over to Amos II and are converted to Amos Exceptions.

**Internal errors** Under heavy workload in very rare cases Bigtable API calls might fail returning an *internal error*. Such errors are not closer specified. Treating these errors as quota failures works well, however.

Unfortunately handling these failures on the server side is not sufficient. When performing
3.3. Architecture

bulk loading operations and heavily utilizing the App Engine, server side failure encoding might fail. Hence the ODDSE client has to be able to verify chunk uploads in order to detect false states. The number of uploaded rows is therefore always compared to the number of actually inserted rows.

3.3.3. The ODDSE client

Figure 3.10.: The interaction among the main components in the ODDSE client. The statistics manager is only required in case of query processing as the bulk file partitioner doesn’t require such statistical data.

The ODDSE client is implemented as an Amos II wrapper in Java. As already mentioned earlier in this chapter the ODDSE client itself consists of four main components. These are the resumable query manager, the query scheduler, the query / data partitioner and the runtime optimizer as illustrated in Figure 3.10. For query processing an additional statistics manager provides histograms to build query chunks each returning a partition of the result of the owner query.

An additional component, the schema manager, maps Bigtable data elements to Amos II elements by transforming Bigtable relation schemas into corresponding Amos II ODDSE
3. ODDSE (Optimized Distributed DataStore Exchange)

schema representations and vice versa.

The following specification will give a more detailed overview about the client’s internal components, their functionality and dependencies. The components presented refer to Java classes of the ODDSE wrapper. To increase readability, compound Java class names are separated by spaces.

**Resumable query manager** The resumable query manager controls processing of queries and bulk upload operations. Queries in terms of the resumable query manager are both GQL queries as well as bulk uploading operations. In order to be fault tolerant and fulfill the basic demand for reliability in ODDSE it implements a resume mechanism allowing to recover from cloud failures and to continue queries.

Currently ODDSE uses the freely available Java library Zql\(^2\) to parse GQL when necessary. The class *query structure* shown in Figure 3.11 encapsulates this library. The demands on SQL parsing are relatively low as Google’s GQL on the server side supports nothing else than simple Select-Project-Queries with some further restrictions.

Every kind of *resumable query manager* provides the two methods *runQuery()* and

runQuery( AppEngine Failure ). These methods enable queries to be stateless regarding execution on the query scheduler. Therewith the same query instance can be simultaneously executed as many times as required. In case of failure an AppEngine failure instance provides the required context to resume the query.

Figure 3.12.: UML class diagram showing Java classes of the client modeling App Engine failure that are used for resumption. All classes provide context information of a previously failed query in order to re-schedule the query on the query scheduler and resume from its point of failure.

Figure 3.12 shows the class hierarchy of AppEngine failures and presents the context information such classes contain. The so called resumePointTuple identifies the last returned data tuple and enables the query to resume from that point. The set of POST parameters (postParams) handed over from the previous query allows furthermore a lazy configuration of the POST method with all necessary parameters. Quota failures provide additional context information as observable from Figure 3.12. The resource identifier is used to recover globally; that means adapting the behavior of all queries in the system accessing the same resource. The necessity to do so becomes obvious when considering that all these queries are consuming quotas of a specific shared resource. If one query runs out of quota all others are almost certain to follow.

Query and data partitioner Query and data partitioners offer chunk generation according to certain system parameters. A query chunk thereby is a query that returns a partition of a given owner query. A data chunk in terms of bulk uploading correspondingly refers to a partition of a given bulk data file. Small chunks in both cases allow massive parallelization utilizing several connections to the App Engine at the same time in order to optimize the system’s throughput. This execution strategy takes advantage of the characteristics of ODDSE’s cloud services, which are designed
3. ODDSE (Optimized Distributed DataStore Exchange)

to scale under heavy load, but allow restricted resource usage only.

Figure 3.13.: UML class diagram of Java client classes showing the partitioners implementing the interface chunk iterator.

As displayed in Figure 3.13 there are three main mechanisms available to build chunks. Two of them serve for query processing and one serves for bulk loading:

- **Partitioners** are available for numerical and string based data types. Either assuming flat distribution or applying a more sophisticated wavelet based histogram partitioners are dependent on previous analyzes of the wrapped external data.

- **Chunk cursor managers** follow another approach as illustrated in Figure 3.14. By means of a chunk cursor the chunk cursor manager iterates over a result set as defined by a GQL query directly on Bigtable. As every iteration step is a particular type of a GQL based query it specializes the GQL query manager. Each iteration step returns a chunk that can be utilized to build a chunk query running on the query scheduler. As these iterations don’t return data from Bigtable, except the chunk boundaries, it takes considerably less time than performing a similar query returning all data.

A constantly increasing parallelism can be achieved as more and more chunks
Figure 3.14.: This figure illustrates the iterative retrieval of chunk boundaries using a chunk cursor as provided by the chunk cursor manager. Chunk boundaries are made available from a chunk queue allowing to speed up query processing on the query scheduler by querying multiple data chunks simultaneously.

are scheduled as soon they become available for execution. Nonetheless this is obviously not as efficient as running chunk queries generated from adequately prepared statistical data. Each chunk query request to the ODDSE server using the chunk cursor approach involves a previous request that returned the appropriate chunk interval. Chunk query generation from statistical data skips this additional server request.

Chunk cursors are independent from the statistical data they provide. They provide an efficient mechanism to run analyze queries for statistics generation.

Bulk file partitioners provide chunk generation for bulk uploading of files containing character separated values, a simple format for data exchange.

Query Scheduler The query scheduler schedules and executes available chunks provided by the resumable query manager. As previously described for Resumable Queries, a query instance provides runQuery methods that can be concurrently run as many times as there are server connections and chunks available.

In order to achieve parallelism queries have to be run in a multithreaded environment using several connections to the App Engine simultaneously. Therefore the scheduler encapsulates the resumable query managers with query threads as illustrated in Figure 3.15. The query threads allow to run chunks simultaneously in parallel.
Figure 3.15.: UML class diagram of client classes illustrating the way resumable query managers are wrapped by query threads to run single chunks on multiple threads using the stateless implementations of runQuery.

Synchronized query threads provide synchronization of asynchronously received partial results, for instance if a sort order in GQL is specified. Synchronization requires an additional helper task to be run. This task, the output synchronizer, is part of the scheduler.

All HTTP connections to the App Engine server are provided using an internal HTTP client\(^3\). The available connections in the HTTP client are shared among all threads leading to resource hazards if not handled carefully. In the worst case a deadlock might occur causing the whole system to fail. Therefore the query scheduler carefully tracks the globally required number of connections, in particular when adjusting this number.

Figure 3.10 illustrates the dependency of the query scheduler from ODDSE’s runtime optimizer. As the availability of an App Engine server varies a lot due to quota consumption the number of connections used has to be constantly adapted. Therefore the current performance is continuously observed by a runtime optimizer and appropriate adaptations are forwarded to the query scheduler in case of changes.

**Runtime optimizer** The runtime optimizer tracks App Engine server status information. This provides a basis for interaction improvements and hence optimizes the overall performance of the system. The tracked status information represents a rather vague remote view on the App Engine as there are no possibilities to query precise

\(^3\)ODDSE uses the Jakarta Commons HttpClient, which is available as part of the Apache HTTP Components
information. Recall that it is based on the following two assumptions:

- A large number of timeouts of App Engine service invocations indicates a high workload on the server side.
- A large number of quota failures indicates the oncoming violation of available quotas on the App Engine server.

Every time an App Engine service invocation fails the runtime optimizer updates its App Engine server status according to the above assumptions.

Based on the current server status the runtime optimizer constantly tries to increase the utility achieved in client-server interaction. Optimization is thereby done in two categories: On the one hand chunk generation done by the query partitioner is optimized in terms of the chunk size, on the other hand the number of simultaneous active connections is optimized for the query scheduler.

Both adjustments are thereby done on two occasions: A resource decrease step in response to an App Engine Failure frees appropriate resources to avoid continuous failures. In adaptive intervals based on a current failure density opponent, resource allocation steps are performed, which optimistically try to increase the applied chunk size or the number of server connections. The adaptive interval ensures to allocate resources only if advisable, considering the failure density. Both optimization steps are supported by a shared recovery manager which loads a fitting set of shared recovery data as illustrated in Figure 3.17.

According to its implementation the runtime optimizer is actually a composite part of a resumable query manager. This is mainly due to the fact that different kind of queries require different optimizers. The lifetime of a optimizer is hence strongly connected to its query. In the following class diagrams the relationship among queries and optimizers is therefore shown as a composition in UML.

The resumable query managers are classified in two main categories: One type of queries serves for query processing, the other one realizes bulk loading. This classification is important for the optimization. According to Figure 3.16 there are two different types of optimizers each serving its own purpose.

The GQL query optimizer attempts to maximize throughput when querying data on the ODDSE server. The system parameters are continuously adapted. On the one
3. ODDSE (Optimized Distributed DataStore Exchange)

Figure 3.16.: UML class diagram of client Java classes presenting the class hierarchy of resumable query managers and runtime optimizers. There are two concrete optimizers, each attached to disjunctive sub trees of the manager hierarchy. The first optimizer serves for bulk loading, the second one for querying.

hand the degree of parallelization when interacting with the server has to meet the available resources. Concurrently querying the server at a high degree having only few resources available will degrade the system’s performance. Most certainly the overwhelming proportion of responses in such case consist of quota failures only. On the other hand an appropriate chunk size has to be chosen. If the chunk size is too big timeout failures are likely to happen. A timeout failure results in a sequential resumption of that query and hence degrades ODDSE’s performance. Choosing too small chunks by contrast requires much more requests to the server, leading to an increasing overhead and the consumption of more resources than necessary. Adaptive interaction optimization for GQL queries is described in Section 5.2.

The bulk loading optimizer tries to maximize throughput when bulk uploading data to the App Engine. Storing data in Bigtable is for more expensive than just querying as it requires write operations. To face this fact a bulk load chunk size is optimized separately from the chunk size optimized by the GQL query optimizer. Additionally Bigtable sub-chunks are introduced reflecting the amount of data sent to Bigtable in one single API call. Similar to previous chunk size considerations, there is an important trade-off between an increasing risk of timeouts and a high consumption of resources. Sending complete App Engine chunks at once to Bigtable might fail due to more restrictive timeouts on the API. Sending single entities by contrast results in unnecessary many API calls causing high resource consumption and large overhead.
Adaptive interaction optimization for bulk uploads is described in Section 4.2.

Runtime optimization depends on the App Engine status retrieved from recent failure information. For this reason App Engine failures are tied closely to the optimizer. In Figure 3.17 all highlighted classes are part of the runtime optimizer. Shared recovery data tracks failure information for a certain type of failure (AppEngine failures) on a specific level. Such levels are either specific queries or ODDSE servers in terms of resources. The following example should help to understand the importance of this distinction:

Two different queries, each split in a number of chunks, are run simultaneously. Both queries access the same ODDSE server. The projection of Query 1 accesses 20 attributes, Query 2 by contrast only returns the key attribute. Assuming similarly sized chunks, returning a chunk for Query 2 should take less time. Returning a chunk of Query 1 might already result in a timeout while Query 2 runs perfectly fine. Query 1 therefore has to be optimized on the query level, that is the size of all following chunks of this query is reduced.

If Query 1 however returns a quota failure the execution of both queries has to be adapted as they are querying the same resource. The ODDSE server doesn’t distinguish between Query 1 or Query 2; The quota is either available or not.

For that reason quota failures are recovered on a resource level such as the total number of server connections used by all queries accessing that particular ODDSE server is decreased.

In the following all shared recovery data observable from Figure 3.17 is explained:

**Recovery interval** This parameter determines a minimum time period between the occurrence of AppEngine failures mapped to the same instance of shared recovery data. All failures occurring within that recovery interval are regarded as one failure occurrence and are consequently processed in terms of recovery just once.

**Recovery time** The occurrence of the last failure in the above sense is recorded as the recovery time.

**Failure count** Keeping track of the number of failures occurred proved to be important. Under presence of a high number of failures additional resource allocation is discouraged. A failure density by contrast based on the number of recent rel-
3. ODDSE (Optimized Distributed DataStore Exchange)

Figure 3.17.: UML class diagram illustrating the Java class hierarchy of App Engine failures and their associated shared recovery data in the ODDSE client. This data is used by the runtime optimizer to optimize the occurrence of multiple failures.

- **Allocation interval**: Similar to the above recovery time a minimum time period is applied to avoid frequent optimistic optimization in terms of allocation of additional resources. Under high failure count additional allocation is performed in much larger intervals, as defined by the graph in Figure 3.18.

- **Optimize time**: This parameter tracks the time of the last optimistic optimization step performed.

- **Block time**: Quota failures require a particular type of shared recovery data, the *adaptive blocker*. This blocker is used to postpone execution of all queries accessing the same resource according to an adaptive *block time*. The block time is defined by a function of *failure density* given in Figure 3.18.

Last but not least Figure 3.19 illustrates the importance of metadata in the ODDSE client. Such metadata is encapsulated by the *execution context* having all ODDSE parameters associated to it. While the query scheduler and the query partitioner (here displayed as *chunk iterator*) either read from the execution context of a query or from the ODDSE parameters directly, the runtime optimizer updates the execution context as well as ODDSE
3.3. Architecture

Figure 3.18.: This chart defines functions of failure count used to adapt the allocation interval for optimistic optimization as well as the adaptive block Time in case of quota failures.

parameters according to the current App Engine status.

3.3.4. The ODDSE server

The ODDSE server consists of a Python application developed to be run on Google’s App Engine.

As illustrated in Figure 3.20 there is usually more than one instance of the ODDSE server running on the App Engine platform. The actual number of instances is managed by the platform depending on ODDSE’s current load and invisible to developers. If load suddenly increases the platform is capable of adding additional resources.

The ODDSE server uses Google’s cloud services Memcache and Bigtable to provide scalable services itself. As server instances might be replicated to meet higher loads, these two services moreover provide the only way to share data among processes. Thinking about the consequences of this replication there is, for instance, obviously no way to physically store data in the file system. Each instance of the server might end up with different versions of files, others missing them entirely.

From Figure 3.20 the App Engine’s storage system Bigtable is immediately recognizable as a distributed storage system. The characteristics of Bigtable have been described earlier in Chapter 2.1.2. Bigtable uses key-range partitioning with the MapReduce framework to achieve load balancing on the storage nodes hosting Bigtable tablets. As ODDSE offers
3. ODDSE (Optimized Distributed DataStore Exchange)

Figure 3.19.: UML class diagram stressing the importance of the execution context in the system, in particular the various ODDSE parameters. As earlier shown in Figure 3.7 most components are dependent on these parameters. The adaptive runtime optimizers continuously update them.

some optimization to improve the efficiency of such load balancing this is a important point to be kept in mind. Promoting the MapReduce efficiency ODDSE benefits from a better load balancing what generally leads to less redundancy. Chapter 4 explains this in detail as it mostly affects bulk loading.

The two Python modules query processing and schema processing presented in Figure 3.21 perform the actual work done by the server. Both modules depend on the Db relation module which provides tools to manage type metadata as well as to access Bigtable. The resumable layer module, visible in the background, is responsible for the reliability. Both query processing and schema processing might fail due to various reasons. This module adds a resumable layer on which server processing can build on. Exceptions are caught by the resumable layer and presented to the client in orde to provide all necessary information for resumption.

The following list of server components (see Figure 3.21) explains in detail how components work and interrelate:

Google webapp framework The Google webapp framework is a rather simple CGI framework in which ODDSE is built. The framework provides easy ways to access the request parameters that encode queries. It takes care of the response generation based on response data written by the processing modules as well as the underlying resumable layer.
3.3. Architecture

Google’s Cloud

ODDSE Server

webapp

framework

Resumable layer

Query processing

ODDSE Server

webapp

framework

Resumable layer

Query processing

ODDSE Server

webapp

framework

Resumable layer

Query processing

Figure 3.20.: This illustration concretizes the cloud view of the ODDSE client and the storage service given in Figure 3.2. The ODDSE server actually consists of several redundant instances running in the cloud on virtual resources. All these instances are accessing the Bigtable storage service. Bigtable itself uses the key-range based partitions, tablets, for load balancing. An additional caching mechanism (Memcache) furthermore provides significant performance increase.

Datastore API The Datastore API provides high level access methods to Bigtable. As earlier described in 3.3.1, in particular in Listing 3.14, Bigtable access through the Datastore API is based on model classes, which are inherited from the class \textit{db.Model}. In the following some App Engine code fragment is presented to illustrate the way the Datastore API is accessed.

Listing 3.19: Datastore API usage in Python (Continuation of Listing 3.14)

```python
# creating a new Person from the model class Person
person = Person( last_name='Smith',
                 first_name='Donal',
                 key_name='820412-646')
```
3. ODDSE (Optimized Distributed DataStore Exchange)

![Diagram of ODDSE Server](image)

**Figure 3.21.** This design chart of the ODDSE server puts focus on the central Python modules and their interaction with the Datastore API and Memcache API.

```python
# store it using the API
person.put()

# retrieve Person based on its key to delete it
person = Person.get_by_key_name('890917-841')
person.delete()

# simple query on Persons using GQL (returning a list of Persons)
persons = Person.gql(WHERE age >= 18)
for person in persons:
    # change each person and store the changes
    person.last_name = "Mr. %s" % p.last_name
    person.put()
```

The officially recommended navigational API usage presented in Listing 3.19 is rather inefficient. Just imagine updating a list of 100 persons. Each put operation hence would require its own API call in order to access Bigtable. During the development of the ODDSE server this paradigm proved to be unacceptable inefficient, of course.

Digging deeper into the Datastore API there is a couple of methods providing a more convenient set oriented access method. `db.Put()` and `db.Delete()` for instance both take a list of Bigtable entities as parameters, however restricted to a specific serialized
representation. The usage of these two functions greatly improved performance and reduced the presence of quota failures due to less API calls.

All in all the Datastore API provides easy and a high level access to Bigtable encouraging the development of data oriented applications. The usage of certain API functions nonetheless has to be carefully evaluated.

**Memcache API** In the distributed environment of the App Engine a Memcache provides the only mean to efficiently share data among processes, most likely to run on different nodes in the cloud. Google’s Memcache is a highly available hash map with a very low response time. The intensive usage of the Memcache service helped in many ways to optimize the ODDSE processing.

Listing 3.20: Memcache API usage

```python
# Optimized loading of a DbRelation using Memcache
entityType = memcache.get("E_%s" % typeName)
if entityType is None:
    # load from Datastore and store in Memcache
    entityType = DbRelation.get_by_key_name(typeName)
    memcache.set("E_%s" % typeName, entityType)
```

**Db relation** The Db relation module defines the DbRelation class that was described earlier in 3.3.1 when presenting the metadata in ODDSE. Instances of DbRelation make up the ODDSE server data dictionary and therewith each describe a Bigtable relation. Listing 3.21 shows a Python code fragment defining the DbRelation class whereas Listing 3.20 presents a short example of its usage.

Listing 3.21: The Datastore model class DbRelation

```python
class DbRelation(db.Model, StaticDbRelationModel):
    """The type DbRelation describes elements of the internal data dictionary used in ODDSE. It supports inheritance and composite keys."""

    # type name
    name = db.StringProperty(required=True)
    # foreign key to parent type
    parentEntity = db.SelfReference()
    # serialized structure information
    structure = db.BlobProperty()
    # flag whether subclasses are present
```

53
The module `Db relation` provides more functionality than just defining the model class. For example, it provides type conversion tools to map Amos II types to Datastore types and backwards. Finally, the `Db relation` module implements the translation of `DbRelations` instances into the Datastore model classes in Python. Especially this last point contributes great flexibility and increased usability compared to the pretty limiting way of hard coded classes in Python.

As server processing relies on the relevant model classes these have to be highly available. To increase performance and to avoid the necessity of loading `DbRelation` instances from Bigtable whenever required, ODDSE hence uses a two level caching mechanism.

When translating `DbRelations` into model classes, dynamic classes are created in Python’s global namespace. This namespace is cached by the App Engine on single nodes in order to minimize loading time for application. This way the ODDSE server is able to reload model classes from the global namespace.

If a required model class can’t be loaded, ODDSE tries to retrieve the corresponding `DbRelation` from the common shared Memcache. This is shown in Listing 3.21. Such an entity has to be translated into a model class before usage. If however even that fails the entity is finally loaded from Bigtable and cached for later usage.

**Resumable layer** This layer is responsible for reliability of an ODDSE server. For any kind of processing the resumable layer is capable of catching exceptions that might occur during processing.

In order to apply the resumable layer to a query the module provides a Python function wrapper `runResumable()`. This function wrapper takes as argument any server side processing function along with an argument list and applies the function in a reliable environment.

**Query processing** According to the ODDSE protocol (see Section 3.3.2) queries are already parsed on the client side and presented as set of POST parameters. These parameters are accessed using the webapp framework.

Processing GQL query strings on the server side didn’t give any increased flexibility, but proved to be inefficient due to expensive string parsing for every single chunk.
Nevertheless this module provides some important tools to improve the handling of
queries split into various chunks. In order to avoid the recreation of a GQL query
object (recall Listing 3.18) for every chunk implying the transmission of all query
parameters, only one query object of the type GQL query is shared among all chunks.
This instance is fast to access through caching. A chunk query only needs to tune
the GQL query object to its relevant boundaries with two additional filters $q.key \geq a$
and $q.key \leq b$.

**Schema processing** This module gives all necessary functionality to exchange schema
data with the client. It supports creation of new Bigtable relations in correspondence
with creation of DbRelations as well as metadata exchange between server and client.
Accessing and changing the ODDSE server schema implies querying and storing
DbRelation instances from the server data dictionary.
4. Bulk loading

Bulk loading or insertion is a common operation for most information based web applications or services. There are several use cases one could imagine:

**Restore** The one use case that stands out is the ability to restore a running system that is already out there in the cloud. Hopefully not being ever used in the vast majority of cases, being able to restore a system from data failures is essential however. Losing all your business data blindly trusting your cloud storage not only might cause severe problems to your business application, but might further even barter away all the trust you and your application gained.

Loosing data due to cloud failures is certainly not likely to happen. Data is stored redundantly and can be easily restored in case of a single node failure [WVLKT08, Wei07]. Clouds however are build on cheap commodity hardware that is sooner or later meant to fail. Even though failure of single nodes is expected there remains a certain risk naively relying on provided redundancy but not being able to control it. [AFG+09] describes the case of the online storage service *The Linkup* that shut down in August 2008 after losing 45% of its customer data. Such cases indicate that relying on one single cloud provider still remains a single point of failure regardless of globally distributed data centers using sophisticated replication mechanisms.

Restoring backup data of a production system at a large scale usually implies insertion of huge amounts of data that might be difficult to handle. ODDSE is chunking such data into small pieces that are uploaded by means of heavy parallelization. In order to optimize throughput chunk generation, as well as the number of client-server connections are continuously adapted.

The ability to restore a system is however meaningless as long as no corresponding scalable backup mechanism is provided. Using the SQL interface of ODDSE such backups can be easily achieved and be stored in any desired format.
ODDSE bulk loading currently focuses on bulk insertion of character separated values. It therewith supports the CVS format (comma separated values), a simple but common data exchange format.

**Initial Upload** Developing a data based service for the web most probably leads to large data volumes already gathered during local development on the App Engine SDK. The amount of your local data might fast exceed the capabilities of Google’s uploading tool. In many cases it is precisely this data that adds value to your application. Uploading the application without it might not make sense at all. Just think of a search engine that is undoubtable depending on a huge data set provided by its crawler.

In contrast to the earlier mentioned obstacle of data lock-in we suddenly experience a lock-out situation. All the valuable data is around, but we are not able to upload and utilize it.

Using ODDSE’s capabilities to bulk load data from local bulk files as described in the previous section, the initial upload of your application data becomes a trivial task in three steps. Query your data from your development server, dump it into the CSV format and finally use the ODDSE bulk loading tool to send it out to the cloud.

The initial upload can be simplified to one single step with only little effort however. All that has to be done is to pass through the already chunked HTTP value stream that ODDSE retrieves from your development server and directly hand it over to the bulk loader tool. As both, the HTTP value stream and the chunk value stream implement the value stream interface they are rather simple to replace.

**Batch uploads** In general frequent updates or inserts of reasonable big data sets are discouraged on production systems in order not to stress their performance under daily use. This is for example a common paradigm for Data Warehouses where updates are run as batch jobs during low use time.

A reasonable example that could be run on top of Google App Engine is a link dictionary requiring constant updates from partners’ data or separate crawling systems. Instead of bothering the system with frequent updates a more efficient way would be a daily batch job bulk inserting all updates of the previous 24 hours at any time the number of users of your system is rather low.
4. Bulk loading

<table>
<thead>
<tr>
<th>Bulk loader set up</th>
<th>Duration</th>
<th>Rows</th>
<th>Throughput</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 connections, chunks of 10 rows</td>
<td>333.7s</td>
<td>6810</td>
<td>3.3 kB/s</td>
<td>failed</td>
</tr>
<tr>
<td>10 connections, chunks of 10 rows</td>
<td>745.3s</td>
<td>14960</td>
<td>3.2 kB/s</td>
<td>failed</td>
</tr>
<tr>
<td>10 connections, chunks of 100 rows</td>
<td>347s</td>
<td>6500</td>
<td>3.0 kB/s</td>
<td>failed</td>
</tr>
<tr>
<td>10 connections, chunks of 100 rows</td>
<td>515.7s</td>
<td>11400</td>
<td>3.5 kB/s</td>
<td>failed</td>
</tr>
</tbody>
</table>

Table 4.1.: The Google bulk loading failure

Due to high computational demands required for write operations the system has to handle bulk loading operations in an efficient way.

For ODDSE this becomes even more crucial as resources are limited by hard, inflexible quotas. Difficulties with bulk loading become even more evident when evaluating Google’s own bulk loading tool, provided as part of the App Engine SDK.

Basis for this evaluation was the dump file used to evaluate ODDSE’s own bulk loading tool. This dump file contains 23.5 MB of customer data being exactly 150.000 tuples. The file was generated using the TPC-H database benchmark, a decision support benchmark system that provides its own data generator. The generated data is chosen to have broad industry wide relevance [Cou99].

As visible from Table 4.1 Google’s bulk loading tool is not able to deal with the challenges of bulk loading data on the App Engine. All test runs by the provided bulk loader were terminated due to Datastore failures. The tool was not able to handle these failures. A manual resumption of the bulk insertion operation is possible as the tool keeps track of successfully uploaded entities. Therefore, a simple script could possibly be used to reload Google’s bulk loader whenever failing due to Datastore exceptions.

However, looking at the efficiency of the tool in terms of data throughput given in Table 4.1 a simple resumption script is not even worth a try. The achieved data throughput of 3.0 to 3.5 kB/s utilizing 10 simultaneous connections to the datastore seems to be rather ridiculous. This throughput was calculated based on the amount of raw data transferred. Log data provided by the bulk loader however states a higher throughput of roughly 8.0 kB/s. This implies a high protocol overhead of more than 100%.

The ODDSE bulk loader was build having essential lessons learned from Google’s bulk loading failure. As earlier described in Section 3.3.2 ODDSE uses a lightweight protocol imposing as little overhead as possible. In order to ensure reliability, App Engine and
Figure 4.1.: This chart shows the difference between fixed and adaptive parameters for bulk loading operations. On the Y-axes the number of successfully inserted rows per request is presented. The total number of rows was in this case always $\approx 65,000$ (10 MB). As observable the overall adaptive yellow query performs best, followed by the adaption of the number of used connections.

Dataport failures have to be handled in such a way that bulk loading operations can be resumed. Using chunks and multiple simultaneous connections is not yet sufficient to scale. Both, the size of chunks and the number of connections have to be adapted to the App Engine’s behavior, in particular in terms of failures.

The ODDSE architecture as already described in Section 3.3.3 includes the query partitioner in charge of chunk generation and the query scheduler which manages the simultaneous connections to the App Engine. The runtime optimizer adapts the behavior of these two components according to an estimated App Engine status depending on the failures observed. With these three components the ODDSE bulk loader is able to meet the previously defined lessons learned from Google’s bulk loader.

Figure 4.1 gives a first impression of ODDSE’s bulk loading approach. The graph illustrates the major impact of adaptive versus static parameter set ups as was learned from the behavior of Google’s bulk loader. Important to notice is that all operations of the ODDSE bulk loader always succeed no matter how bad the set up is. The bulk insert presented last in Table 4.2 already outruns Google’s tool by a Factor of more than 10.
4. Bulk loading

<table>
<thead>
<tr>
<th>ODDSE set up</th>
<th>Duration</th>
<th>Rows</th>
<th>Throughput</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>static chunks, sequential</td>
<td>1421,6s</td>
<td>64373</td>
<td>7,2 kB/s</td>
<td>succeeded</td>
</tr>
<tr>
<td>adaptive chunks, sequential</td>
<td>1231,4s</td>
<td>64373</td>
<td>8,3 kB/s</td>
<td>succeeded</td>
</tr>
<tr>
<td>static chunks, adaptive connections</td>
<td>528,4s</td>
<td>64373</td>
<td>19,4 kB/s</td>
<td>succeeded</td>
</tr>
<tr>
<td>adaptive chunks and connections</td>
<td>275,1s</td>
<td>64373</td>
<td>37,2 kB/s</td>
<td>succeeded</td>
</tr>
</tbody>
</table>

Table 4.2.: Evaluation of Figure 4.1 (ODDSE bulk loader)

The performance of the ODDSE uploader is thoroughly investigated in Section 4.3 and even better results are presented.

In the following the bulk loading tool and its components are described in detail and finally evaluated in Section 4.3.

4.1. Chunk generation

Figure 4.1 showed the importance of an adaptive chunk generation to achieve higher performance. Allowing adaptive chunks requires a flexible but efficient tool to split large files into small chunks. In particular when using multiple connections this becomes a none trivial task.

4.1.1. Sequential approach

In order to have a reference point for later optimizations a sequential chunk generator was implemented first. Using a LineNumberReader in Java this is more or less a trivial task and therefore not thoroughly explained here. The concept is rather simple:

Whenever the single server connection is available a specific number of rows is read from the data file, transformed according to the protocol and uploaded to the server. The server’s response is analyzed as soon as received. If an App Engine failure is indicated and/or not all rows are inserted, successfully appropriate actions are taken. Due to the occurrence of an App Engine failure the ODDSE runtime optimizer first updates the App Engine status information it keeps track of. Afterwards all missing rows are resent to the App Engine server.

As soon as a chunk is successfully uploaded the partitioner, working in a sequential manner,
4.1. Chunk generation

<table>
<thead>
<tr>
<th>ODDSE set up</th>
<th>Duration</th>
<th>Avg. latency</th>
<th>Summed latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>static chunks, sequential</td>
<td>$1421,6s$</td>
<td>$2,2s$</td>
<td>$1412s$</td>
</tr>
<tr>
<td>adaptive chunks, sequential</td>
<td>$1231,4s$</td>
<td>$2,8s$</td>
<td>$1225s$</td>
</tr>
</tbody>
</table>

Table 4.3.: Evaluation of Figure 4.1 (Latency for sequential bulk uploads)

reads the succeeding rows from the data file. The number of rows to read is determined by the runtime optimizer depending on the current App Engine status.

In order to bridge the high latency of the server response and the associated required idle time, pipelining can be used to speed things up. As soon as a request is send, the ODDSE client already prepares the following chunk. This however delays takeover of an adapted chunk size for following chunk. As chunks are generally very small the achieved speedup is very little. Nevertheless such preparation of chunks in advance became important when introducing multiple simultaneous connections.

As the sequential approach obviously doesn’t perform great a sequential partitioner is not available in the system anymore. Instead this partitioner was developed further to enable parallelism.

4.1.2. Parallel approach

ODDSE faces high idle times due to high latency of server responses when using a sequential approach as illustrated in Figure 4.2. Table 4.3 gives a short evaluation of the latency of both queries presented in Figure 4.2. More than 99% of the processing time is spend idly waiting for the servers’ response. To improve performance multiple client-server connections hence should be used simultaneously. For that reason ODDSE uses a pool of threads having each thread provided with its own server connection. Increasing the workload on the server due to additional requests most likely increases the latency even further. The impact of latency is reduced by a factor equivalent to the number of additional resources.

The bulk loading manager supports the usage of simultaneous connections to the ODDSE server. This requires the ability to load chunks on the client simultaneously, which imposes some more requirements on the chunk generator. This chunk generator is a bulk file partitioner as presented in Figure 3.13. As several threads, each of them associated to
Figure 4.2.: This graph presents the latency measured when processing the sequential bulk uploads from Figure 4.1. One can observe an increasing latency for the adaptive query due to growing chunks.

A server connection, request data chunks simultaneously operations have to be thread safe. In order to keep threads from waiting to access synchronized methods a chunk is automatically generated whenever a chunk task is scheduled by the query scheduler. Scheduling of chunk tasks is done sequentially by the main thread during idle times. This helps to evade unnecessary mutual blocks.

Generated chunks are available from a chunk queue. This queue ensures that always enough chunks are available whenever the query scheduler runs a scheduled chunk task. The ODDSE main thread on the other side ensures the availability of chunk tasks that are ready to be run and are being scheduled. Therewith new chunk tasks can be run efficiently whenever a thread finishes processing its current task and the associated server connection is released. Switching time of tasks is reduced to a minimum.

Pre-preparation of chunks has its drawback, however. The more chunks are prepared at an earlier stage, the less impact can be achieved using an adaptive chunk size. At adaption time chunks are already created and available from a chunk queue. To minimize
4.1. Chunk generation

Figure 4.3.: UML class diagram presenting different types of value streams in the ODDSE client used for bulk loading and query processing. Due to the common abstract superclass a HTTP value stream could be used to bulk upload a queried data set without much effort as well.

this drawback the size of the chunk queue is restricted and allows a maximum of one prepared chunk per available server connection. This ensures a fast reuse of available server connections, even in the worst case having all connections freed at the same time. On the other hand chunk size adaptations are realized almost immediately.

Whenever the chunk queue is filled up to its extent all attempts to add further chunks are blocked. At the same time scheduling of new chunk tasks is postponed.

To minimize main memory usage in the ODDSE client the chunk content is not loaded into memory on chunk generation until a data chunk is run by the query scheduler. At that point a chunk value stream is opened, using the specialized chunked file input stream as presented in Figure 4.3. Chunked file input streams give an easy way to access bulk files split into various chunks.

The size of chunks determined by the runtime optimizer is given using rows as unit whereas chunked file input streams expect the size to be in Bytes. Optimizing chunks in terms of rows is anyhow more useful. First of all it enables one common model for select queries and bulk loading operations. Actual costs on the App Engine server are furthermore proportional to the number of rows rather than the size of data.

To convert a couple of rows given by the runtime optimizer to the approximate equivalent in Bytes the average size of rows is tracked by the bulk file partitioner and updated as long as necessary to provide reasonable results.
4. Bulk loading

The chunked file input stream extends the capabilities of Java’s file input stream. Such a chunked input stream not only takes a file name as argument on initialization, but moreover an offset and a size both given in Bytes. It will actually return the first complete row after the given offset and correspondingly stops streaming after returning the last row that was partially started within the range defined by offset and size. Such a chunk definition in terms of offset and size avoids loading chunk content into memory and helps to save memory resources.

In order to provide a common model for input and output of data ODDSE internally uses value streams as presented in Figure 4.3. While queries return a HTTP value stream bulk uploads currently use a chunk value stream that encapsulates a chunked file input stream. This architecture gives ODDSE the flexibility to upload data from various sources requiring only few additional implementation work.

4.1.3. Randomized parallel approach

Parallelizing the chunk upload still doesn’t solve the obstacle of high average latencies. The impact however was reduced to a minimum.

[SCS+08] studies the problem of bulk loading records into distributed ordered tables which are range-partitioned according to their key over a large cluster of shared nothing machines. Such distributed storage systems are for example Yahoo PNUTS [CRS+08] or Google’s Bigtable [CDG+06] which is wrapped by ODDSE in this project. [SCS+08] proposes a bulk insertion framework based on three phases: staging of data, planning for the bulk insertion and finally the actual insertion operations. The proposed framework however relies on internal specifics such as partition sizes, the number of partitions, the way partitions are split and redistributed and several more. Hence the framework is not applicable for ODDSE as Bigtable partitions are transparent to us. Nevertheless, [SCS+08] evaluates its proposed framework against a randomized bulk loading approach. The randomized insertion of chunks already improved the bulk loading operation according to [SCS+08] by a factor of 2.

How is that speedup achieved? The difficulty of bulk loading results from key-sorted bulk data. When uploading chunks in such order the operation often results in one hot partition. The key range of a hot partition then embraces the keys of all simultaneously uploaded chunks. Instead of balancing work load based on horizontally partitioned and
Figure 4.4.: This chart compares the latency measured when performing sequential bulk
loading operations in contrast to operations utilizing multiple simultaneous
connections to an App Engine server. One can observe that the measured
latency differs by a factor of nearly 2.5.

distributed data, all workload resides on just one single storage node. This decreases
the expected impact of parallelism using multiple simultaneous connections to the server.
The more connections are used at the same time the higher the workload gets on a hot
partition. As a result latency increases enormously impairing performance of the bulk
loading operation.

Figure 4.4 compares the latency of two bulk loading operations, both having static chunk
sizes in order to be comparable. The sequential operation shows a much lower latency than
the one using multiple adaptive connections. In average the latency of parallel operations
is by a factor of 2.5 higher than for sequential operations.

In the App Engine’s Datastore tables are range-partitioned into several tablets as presented
in Figure 4.5. Each tablet is stored on a different storage node and about 100 to 200 MB
in size [CDG+06]. Whenever setting up a new table it is located on just on storage node
for the moment. With growing size the start tablet is split into more tablets which are
moved to other storage nodes. Whenever data is requested from the Datastore or inserted
into it, the workload is divided between several storage nodes - assuming however that
4. Bulk loading

Figure 4.5.: Every table in Bigtable consists of one or more tablets containing key-range partitioned data [CDG+06]. Such tablets are approximately 100-200 MB in size. With growing size tables are automatically split.

not always the same entities or key ranges are accessed. Generally this mechanism helps to scale the database and allows huge load.

When bulk loading key-ordered data however this assumption fails. As described above it leads to hot partitions having all workload accumulated on just one Bigtable tablet node instead of various as shown in Figure 4.5. High latency (see Figure 4.4) and even a raise of timeout failures was observed.

The idea of randomizing uploaded chunks aims at the prevention of hot partitions trying to utilize all available tablets randomly. This however requires several tablets to exist. As tablets initially consist of one tablet only, obviously no speedup can be achieved for initial uploads.

A reasonable large data set however is likely to be located on two or more storage nodes. Randomizing chunks of later updates or inserts could help to support load balancing among storage nodes as it hopefully affects them randomly. In order to evaluate the impact of such a randomized approach ODDSE provides a randomized bulk loading tool.

Instead of restricting the size of the chunk queue as done previously, the randomized bulkloading manager prepares all chunks before a connection to the App Engine server is
opened. Afterwards chunks are chosen randomly from the chunk queue and uploaded to the Datastore.

One implementation characteristics helped to achieve this more or less memory efficient. Loading the content of all chunks into memory is certainly impossible or at least discouraged for large data sets. As mentioned earlier each data chunk can be described by offset and size allowing to open chunk value streams precisely when required on the query scheduler on execution time. Preparation time of chunk value streams is besides rather non-significant to the overall duration.

However there are still two major drawbacks of the randomized approach. First of all even using a minimal description of data chunks by offset and size, the preparation of all such chunks in advance remains memory costly. But moreover all potential to adapt chunks to a changing App Engine status is lost.

### 4.2. Runtime optimization

Optimization in ODDSE is crucial in order to ensure scalability, reliability and high performance. Not carefully observing the App Engine’s status results in a rapidly increasing risk of App Engine failures. The higher the overall workload is the more uncontrollable this risk gets.

Even though ODDSE is able to overcome App Engine failures resulting from timeouts, quota violations and several more reasons continued unadapted requests to the server in such situations might finally lead to a unrecoverable server state. In particular in terms of bulk loading of huge data files for instance this was observable. Optimizing the server-client interaction however keeps the risk of failures to a reasonable level and allows ODDSE to scale in terms of huge amounts of data considering reliability and performance.

Splitting and redistributing tablets (see Figure 4.5) during bulk insertion, for instance, might be phases when Bigtable storage nodes become more sensitive to additional high workload, leading to internal failures. Such issues however can be addressed using a dynamic interaction model that is sensitive to the servers status.

At the same time failures degrade the systems performance due to expensive resume operations. Any failed chunk request requires resume actions in the client. After client resumption all data has to be resend to the server what involves high latency. If moreover
the data allocated to a failed chunk exceeds the current capabilities of the App Engine server the chunk has to be resplit into smaller chunks. These chunks are then processed sequentially using the one server connection that was allocated to their root chunk. Failures are therefore one of the biggest performance obstacles observed in the system. To assure a reasonable throughput, when bulk loading entries from data files, optimization has to be carefully done to minimize the occurrence of failures.

Very small chunks considerably reduce the risk of timeout failures. On the other hand such small chunks show a larger overhead regarding latency and quota usage of the Datastore API. A resembling trade-off was experienced for the number of simultaneous connections based on the effect of increasing latency as shown in Figure 4.4.

Analyzes done by the runtime optimizer enable constant improvement of the ODDSE interaction with the App Engine server. This way ODDSE can meet the highly dynamic demands in communication due to a changing App Engine status.

The runtime optimizer establishes its analyzes upon a continuous tracking of the App Engine status. As there is no way to query the actual status of an App Engine server, ODDSE uses some assumptions to appraise current status information. An increasing number of timeouts of App Engine service invocations indicates a high workload on the server side. An increasing occurrence of quota failures on the other side predicts the oncoming violation of available quotas on the App Engine server.

On failures the runtime optimizer updates the App Engine server status according to the above assumptions. Centric part of the server status is a logarithmical failure density for all known failure types, calculated on bases of a failure counter:

\[ \text{density}_b(failure) := \log_b(b + \text{count}_{\text{now}}(failure)) \]

whereby:

\[
\text{count}_{t+1}(failure) = \begin{cases} 
\text{count}_t(failure) + 1 & \text{if failure decrease step (failure)} \\
\text{count}_t(failure) \times 0.9 & \text{if resource allocation step (failure)} \end{cases} \wedge \\
failure \in \{\text{TimeoutFailure, QuotaFailure, CursorFailure}\}
\]

As expected \(\text{count}_t(failure)\) is increased whenever \(failure\) occurred and a corresponding decrease step is performed. In case of a optimistic resource allocation step however the counter is decreased by 10%. The interval of resource allocation steps is thereby dependent
4.2. Runtime optimization

on the counter itself. This way the counter provides a number of relevant recent failures. The logarithmic derivation of the failure density $density_{failure}$ from $count_{failure}$ allows adaption according to recent failures but restricts the impact of permanent failures at the same time.

Optimization by the runtime optimizer is done on two different occasions in opponent steps:

**Resource allocation step** Whenever the query scheduler is about to execute a chunk task the runtime optimizer evaluates whether to perform resource allocation steps. Resource allocation is always done regarding a specific failure type as different failures relate to different resources. In order to perform a resource allocation step the following condition has to hold:

$$t_{\text{now}} - \max(t_{\text{allocation}}(failure), t_{\text{recovery}}(failure)) > I_{\text{allocation}}(failure)$$

whereby:

$$I_{\text{allocation}}(failure) = I_{\text{defaultAllocation}} \ast \text{density}_2(failure)$$

The above condition ensures to allocate the less additional resources regarding a certain type of failure the higher the density of this failure is. This is achieved by increasing the minimum allocation interval $I_{\text{allocation}}(failure)$ that has to pass after a earlier allocation step $t_{\text{allocation}}(failure)$ regarding the failure or any occurrence $t_{\text{recovery}}(failure)$ of it. Figure 4.8 illustrates $I_{\text{allocation}}(failure)$ depending on the number of recently observed failures of type failure.

A resource allocation step regarding timeouts will generally increase the size of chunks. Regarding quota failures on the other side the number of connections will be increased.

**Failure decrease step** In case of a failure the runtime optimizer checks whether to perform a failure decrease step freeing allocated resources. This is once again done depending on the failure type.

As before the step requires a certain minimum interval to pass after the last decrease step $t_{\text{recovery}}(failure)$. This interval $I_{\text{defaultRecovery}}$ is not adaptive, but static and should be set close to the average latency observed in the system. Therewith the effect of a certain parameter set is approximately restricted to just one failure decrease step.
4. Bulk loading

A failure decrease step regarding timeouts will decrease the chunk size in order to avoid such a failure. Regarding quota failures on the other hand the number of simultaneous server-connections is reduced.

Parameters are adapted during resource allocation steps as well as failure decrease steps. Fundamental part of all incremental (+) and declining (−) adaptions is the sigmoid function \( \text{adapt}(x, \text{failure}) \):

\[
\text{adapt}(x, \text{failure}) := \frac{f_{\text{max}} + f_{\text{min}}}{2} \pm \frac{f_{\text{max}} - f_{\text{min}}}{2} \star \frac{h(x, \text{failure})}{\sqrt{1 + h(x, \text{failure})^2}}
\]

whereby:

\[
h(x, \text{failure}) := \frac{s}{\text{density}_{10}(\text{failure})} \star (x_{\text{opt}} - x - \frac{1}{s} - \log_2(x_{\text{opt}}) \star \text{density}_2(\text{failure}))
\]

The above adaption function relies on certain important parameters. \(x_{\text{opt}}\) is the target value \( \text{adapt}(x, \text{failure}) \) is adapting towards. In case of incremental adaption \((x \star \text{adapt}(x, \text{failure}))'\) is approximately \(f_{\text{max}}\) for \(x \ll x_{\text{opt}}\) and \(f_{\text{min}}\) for \(x \gg x_{\text{opt}}\). For declining adaption the opposite applies to \((\text{adapt}(x, \text{failure}))'\).

The turning point of the sigmoid function \( \text{adapt}(x, \text{failure}) \) is at \(x_{\text{opt}} - \frac{1}{s} - \log_2(x_{\text{opt}}) \star \text{density}_2(\text{failure})\). \(s\) represents a scaling factor determining the gradient in the turning point. Moving the turning point by \(\frac{1}{s}\) towards the origin gives more adequate results for \(x \to x_{\text{opt}}\) that are closer to \(f_{\text{min}}\). If failures are present \(\log_2(x_{\text{opt}}) \star \text{density}_2(\text{failure}) \neq 0\) further moves the turning point towards the origin as shown in Figure 4.6. \(\log_2(x_{\text{opt}})\) thereby scales the failure density. The illustrated inclusion of the failure density reduces the target value \(x_{\text{opt}}\) to \(x_{\text{opt}} - \log_2(x_{\text{opt}}) \star \text{density}_2(\text{failure})\) leading to a desired lower incremental but higher declining adaption.

The parameters in the following list are most relevant for bulk loading operations. Their impact and optimistic incremental adaption, as well as failure caused declining adaption will be described here.

**Simultaneous connections (query scheduler)** The usage of simultaneous connections helps to increase data throughput as earlier described relating to Table 4.3. As shown in Figure 4.4 it considerably increases latency at the same time however. This implies that all computations get closer to their timeout limits hence increasing the occurrence of timeouts. Based on this parameter ODDSE can leverage between both sides and deal with the trade-off more efficiently.
This parameter is closely connected to quota failures and all adaption is hence only done regarding such failures. Adapting the size of chunks contrary due to the lesson learned from Figure 4.4 was avoided in order to keep the adaption dependencies more simple.

The number of simultaneous connections $\text{conns}$ is adapted by means of the sigmoid function $\text{adapt}(x, \text{failure})$ from above. Incremental and declining adaption use both their own parameter set:

\[
\begin{align*}
\text{adapt}_{\text{dec}}(\text{conns}, \text{failure}) & : \text{conns}_{opt} = 25; f_{min} = 1.01; f_{max} = 1.15; s = \frac{1}{4} \\
\text{adapt}_{\text{inc}}(\text{conns}, \text{failure}) & : \text{conns}_{opt} = 25; f_{min} = 1.005; f_{max} = 1.10; s = \frac{1}{4}
\end{align*}
\]

The target number of 25 connections as $x_{opt}$ is a primitive heuristic showing good results. Comparing the incremental and the declining set-up the latter one is more aggressive leading to a faster decline as also illustrated in Figure 4.6.

![Graph](image)

**Figure 4.6.** The figure presents $\text{adapt}_{\text{inc}}(\text{conns}, \text{QuotaFailure})$ and $\text{adapt}_{\text{dec}}(\text{conns}, \text{QuotaFailure})$ used to adjust the number of simultaneous connections in ODDSE. As visible from $f_{max}$ incremental adjustments are more conservative than declines. Under presence of failures the turning point moves towards the origin as illustrated by the dotted charts.

**App Engine chunks (query partitioner)** The second outstanding parameter after the number of connections applied is the size of chunks uploaded to the ODDSE server. The
4. **Bulk loading**

unit of this parameter is rows instead of the actual amount of data. The reason becomes obvious when considering that data is written per entity, i.e. per row, and therewith determining the number write operations.

Adapting the chunk size is based on the following trade-off: Small chunks considerably reduce the risk of timeout failures but show a high summed latency. Big chunks on the other side are much more likely to fail due to timeouts but have less overhead in terms of overall latency and the number of API calls.

This parameter is closely connected to timeouts and all adaption is hence only done regarding such failures.

The chunk size is adapted by means of the sigmoid function \( \text{adapt}(x, \text{failure}) \) from above similar to the adjustment of connections. Incremental and declining adaption use both their own parameter set:

\[
\text{adapt}_{\text{dec}}(\text{rows}, \text{failure}) : \text{rows}_{\text{opt}} = \text{rows}_{\text{avg}}(t_{\text{now}}); f_{\text{min}} = 1.005; f_{\text{max}} = 1.15; s = \frac{1}{20}
\]

\[
\text{adapt}_{\text{inc}}(\text{rows}, \text{failure}) : \text{rows}_{\text{opt}} = \text{rows}_{\text{avg}}(t_{\text{now}}); f_{\text{min}} = 1.005; f_{\text{max}} = 1.10; s = \frac{1}{20}
\]

whereby:

\[
\text{rows}_{\text{avg}}(t) := \frac{\sum_{i=1}^{5000} \text{rows}(t - i)}{5000}
\]

The target size \( \text{rows}_{\text{opt}} \) refers to a floating average of the last thousands successfully inserted rows per chunk. Similar to the previous parameter a decline is more aggressive compared to a conservative increase in size. Figure 4.7 illustrates the adaption functions. Compared to Figure 4.6 the curves are less steep due to a much lower \( s \).

**Bigtable sub-chunks (query partitioner)** For expensive insert operations it showed to be advantageously to allow further granularity when accessing the Datastore API as a trade-off becomes evident:

Sending all chunk data to the API at once results in an increasing probability of timeout failures due to more restrictive API operations. Sending each entity in a single call however, requires more API calls what affects the API’s quotas at a higher degree. Chunking of chunks offers higher granularity and helps to meet this trade-off.

Another advantage has to be mentioned here. Even if chunk processing fails, it is in most cases not necessary to upload all data from the client again as some sub-chunks can still succeed.
4.2. Runtime optimization

Figure 4.7.: The figure presents $\text{adapt}_{\text{inc}}(\text{rows}, \text{TimeoutFailure})$ and $\text{adapt}_{\text{dec}}(\text{rows}, \text{TimeoutFailure})$ used to adjust the size of chunks in ODDSE. A lower parameter $s$ gives a softer transition from $f_{\text{max}}$ to $f_{\text{min}}$ and opposite.

To adapt the number of sub-chunks $n \in [1, 10]$ a simple model was sufficient. During a failure decrease step regarding a timeout failure the number of sub-chunks is increased by 20%. Hence the workload on each call to the Datastore API is reduced. When performing a failure decrease step regarding a quota failure the number is decreased by 10% to reduce the overhead due to several API calls. The same is done when running a resource allocation step regarding timeouts.

**Block time (query scheduler)** Quotas on the App Engine server are reset every 24 hours. Per-minute quotas moderate how quickly daily resources can be consumed [Goo08]. Whenever ODDSE is running out of quotas this is usually due to per-minute quotas. A short recovery interval during which all execution of chunks on the query scheduler is blocked allows to regain quotas and continue processing.

There is no way however to automatically detect the violation of storage quotas. In such case the bulk loading operation cannot be resumed until additional quotas are purchased or storage space is freed.

The adaptive block time is managed as part of the shared recovery data that was
4. Bulk loading

... described in Section 3.3.3 and calculated similar to $I_{allocation}(failure)$:

$$I_{block}(QuotaFailure) := I_{defaultBlock} \cdot density_e(QuotaFailure)$$

![Graph showing resource allocation interval and adaptive block time (Quota Failures)](image)

Figure 4.8.: This figure presents $I_{allocation}(failure)$ and $I_{block}(QuotaFailure)$ that are both based on the $count_t(failure)$ respectively $density_e(failure)$.

The set of the presented parameters is quite similar to parameters used to optimize ODDSE queries as described in Chapter 5. The impact of all parameters however is much higher and demands more careful adaption. *Bigtable sub-chunks* is the only parameter which is not present in the query runtime optimizer. The fine granulation it enables is just required for writes to the Datastore.

### 4.3. Evaluation

In this section ODDSE’s bulk loading capabilities are evaluated regarding performance, reliability and scalability.

Basis for the evaluations presented in this sections are several dump files containing approximately between 65,000 to 1,500,000 records, respectively between 10 to 170 MB in size. The files were generated using the TPC-H database benchmark, an ad-hoc decision support benchmark system providing its own data generator. The generated data is chosen to have broad industry wide relevance [Cou99].
4.3. Evaluation

Figure 4.9.: This chart is intended to show the difference among fixed and adaptive parameters. On the Y-axes the number of successfully inserted rows per request is presented. The total number of rows was always \( \approx 65,000 \) (10 MB). As observable the overall adaptive yellow query performs best, followed by the adaption of the number of used connections.

4.3.1. Performance

As already earlier mentioned a huge speedup is achieved applying several server-connections simultaneously. Table 4.4 evaluates the four bulk loading operations presented in Figure 4.9. The total number of rows of these bulk insertions was \( \approx 65,000 \), respectively 10 MB. Between the naive unadaptive sequential bulk insertion and a optimized insertion a speedup of a factor \( > 5 \) could be achieved. The highest average throughput achieved in the presented test case was 37.2 kB/s.

The worst set up evaluated and presented in Table 4.4, a sequential upload with static chunk size, still performs more than twice as good comparing its throughput with the best run of Google’s bulk loading tool. Google’s bulk loader hardly reached a throughput of 3.5 kB/s as earlier presented in Table 4.1. While the ODDSE bulk insert runs sequentially in the referred case Google’s tool already utilizes 10 simultaneous connections. The overall
4. Bulk loading

<table>
<thead>
<tr>
<th>ODDSE set up</th>
<th>Duration</th>
<th>Rows</th>
<th>Throughput</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>static chunks, sequential</td>
<td>1,421.6s</td>
<td>64,373</td>
<td>7.2 kB/s</td>
<td>succeeded</td>
</tr>
<tr>
<td>adaptive chunks, sequential</td>
<td>1,231.4s</td>
<td>64,373</td>
<td>8.3 kB/s</td>
<td>succeeded</td>
</tr>
<tr>
<td>static chunks, adaptive connections</td>
<td>528.4s</td>
<td>64,373</td>
<td>19.4 kB/s</td>
<td>succeeded</td>
</tr>
<tr>
<td>adaptive chunks and connections</td>
<td>275.1s</td>
<td>64,373</td>
<td>37.2 kB/s</td>
<td>succeeded</td>
</tr>
</tbody>
</table>

Table 4.4.: Evaluation of Figure 4.9 (ODDSE bulk loader)

Adaptive bulk upload presented last in Table 4.4 outruns Google’s tool by a Factor of 10 and above.

All bulk operations presented in Figure 4.9 were issued in sequence on one ODDSE server according to the order given in the figure’s legend. Due to quota restrictions on the server this however degrades performance of later operations. The impact of adaptive setups as already observable can hence regarded to be even higher when performing independent runs.

Later in this section more bulk loading setups are evaluated in terms of their parameter adaption. The bulk uploads presented there show further improved performance.

Figure 4.10 shows the number of simultaneous connections applied for the bulk insertions from Figure 4.9 which use adaptive connections. Starting with the same degree of parallelism the bulk insertion using adaptive chunks ends up being faster and using more connections. This supports the assumption that a changing number of connections should be followed by an adequate adaption of the size of chunks as well. For higher numbers of connections smaller chunks are usually better suited as they avoid timeout failures when facing an increased risk for timeouts.

In the following a larger TPC-H dataset was bulk inserted on different App Engine servers to compare to performance of various setups and to evaluate their parameter adjustment. The dataset contains exactly 150,000 rows and is about 23.4 MB in size.

Table 4.5 presents the performance of the bulk insertions illustrated in Figure 4.11. Compared to Table 4.4 obviously a much higher data throughput was achieved. Running bulk loads in sequence certainly affected the result of the previous set-up. Nevertheless the test runs presented in Table Figure 4.5 perform all very well in comparison. For each bulk insertion an insisting greedy resource allocation was used for runtime optimization.
4.3. Evaluation

Figure 4.10.: This chart provides details for the previously shown bulk insertions from Figure 4.9 in terms of the number of simultaneous connections used. Even though the optimization algorithm used to adapt the number of connections is an early and rather non-optimal version, one can still observe that adaptive chunks allow a higher optimal degree of parallelization generally applying smaller chunks as visible from Figure 4.9.

<table>
<thead>
<tr>
<th>ODDSE set up</th>
<th>Duration</th>
<th>Transferred rows</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequential</td>
<td>1,844.8s</td>
<td>150,000</td>
<td>12.9 kB/s</td>
</tr>
<tr>
<td>multiple static</td>
<td>480s</td>
<td>150,000</td>
<td>49.85 kB/s</td>
</tr>
<tr>
<td>connections (static)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>multiple adaptive</td>
<td>452.4s</td>
<td>150,000</td>
<td>52.9 kB/s</td>
</tr>
<tr>
<td>connections (adaptive)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.5.: Evaluation of Figure 4.11 (ODDSE bulk loader)

Figure 4.12 presents details of the query performing best in this ‘greedy’ test run. As observable from that graph the number of applied connections as well as the chunk size was highly and continuing increased in the first part of the loading process. Towards the end when lacking resources especially the number of applied connections plunges from approximately 30 to less than 5. At the same the chunk size declines from 115 to 60.

The presented insisting greedy optimization however is not doable anymore due to changes on the App Engine. At first Google managed quotas using a sliding window of 24 hours. This encourages a greedy approach taking all resources you get during the next 24 hours. The drawback however stands out: Having consumed all resources at once no further operations are possible. The 24h sliding window was replaced by an easier to understand
4. Bulk loading

Figure 4.11.: This graph compares three bulk insertions of 150,000 records each using a rather greedy set-up. The chunks size is in all cases adapted during loading. Due to huge differences in their size the number of inserted rows is presented on a logarithmical scale.

A quota management system. Currently quotas on the App Engine are reset every 24h. But at the same time quota consumption is restricted by per-minute quotas in order to avoid irregular peaks in consumption\(^1\).

In contrast to greedy optimization Figure 4.13 presents a significantly more moderate optimization still showing good results. Both the size of uploaded chunks as well as the number of simultaneous connections used are altered in much smaller ranges. The chunk size reaches a maximum of 75 rows per chunk and never drops under its initial value of 50 rows. The number of applied connections on the other hand reaches shortly 22 and ends up being rather constant between 15 and 11. Bulk insertion of 150,000 records succeeded within 512 secs and an average data throughput of 46.7 kB/s. This data throughput showed to be a good medium result that can be realized using appropriate initial parameters.

Figure 4.14 evaluates the impact of non-optimal initial parameters both for the chunk size as well as the number of server-connections applied. The bulk load operation is initiated with the lowest parameters allowed in ODDSE. This initial set-up is 20 rows

---

\(^1\)Check the Google App Engine documentation for details considering quotas: 
http://code.google.com/appengine/docs/quotas.html
4.3. Evaluation

Figure 4.12.: This graph presents details of the top-runner of Figure 4.11 realizing a throughput of 52.9 kB/s. Even though it uses a rather insisting greedy parameter adjustment for runtime optimization it performs extraordinarily well.

Figure 4.13.: This graph presents a medium result using moderate optimization. The bulk loading of 150,000 records succeeded in 512s and an average throughput of 46.7 kB/s.

and 3 simultaneous connections to the ODDSE server. Bulk insertion of the same bulk file only took insignificant more time. Instead of the previously achieved 512 secs the
Figure 4.14.: This figure evaluates the impact of non-optimal initial parameters for the bulk load chunk size and the number of simultaneous connections. The optimization algorithm is similar to the one applied in Figure 4.13.

4.3.2. Reliability

During the evaluation of bulk loading operations invalid results were noticed in some cases. Even though the App Engine server didn’t return any failure, result rows were missing in the end. This was observable when continuously pressuring the App Engine server with a high number of simultaneous connections and several bulk loading operations in sequence.

It looks like the server’s resumable layer can fail in cases when server processing is interrupted before the actual failure information is encoded and added to the response message.

In order to avoid such unreliable results the client checks any response message for missing failure information. If the number of added rows doesn’t match the number of inserted rows chunk processing on the server side failed and has to be redone.

Being tolerant to server failures as just described ever since, timeout failures, quota failures and last but not least several unspecified failures of the Datastore classified as Internal Errors ODDSE provides a reliable system.
4.3.3. Scalability

Bulk loading of up to 150,000 rows was already earlier shown to succeed. In order to evaluate the limits of ODDSE in terms of scalability a dataset of 1,500,000 rows (170 MB) generated by TPC-H was used. This dataset was bulk loaded to empty Datastores on different App Engine servers.

The bulk operations using the huge TPC-H file remarkably did not fail, but however blocked between 44% to 50% after having inserted approximately 680,000 to 770,000 records.

In all cases the bulk loaded data in combination with all standardized automatically generated indexes soon exceeded the overall available storage of 1GB in the Datastore. The announced free storage quota of 1 GB hence as to be regarded critical. Due to obviously huge storage intensive indexes a much lower amount of storage on the App Engine is actually available free of charge.

As soon as the storage limit of 1 GB was violated no additional inserts were possible. Storage usage however can only be observed from the Google App Engine administration panel. ODDSE itself is unaware of the fact of running out of storage space. Purchasing additional storage quotas from Google or freeing resources any other way the blocked bulk loading operation would simply resume and continue the processing of remaining rows.

Figure 4.15 presents one of the attempts to bulk load the 1.5 million TPC-H records. This attempt failed after inserting approximately 680,000 rows. Analyzing the graph both parameter plots, the chunk size as well as the number of connections, look extraordinary steep and hilly. The optimization algorithm applied however is the same used earlier for instance in Figure 4.13. The reason is rather the huge tick interval of hours in contrast to minutes as earlier used. The algorithm develops towards a parameter set using the maximum number of 50 server-connections together with a minimum chunk size of 20 rows as currently configured in ODDSE. An interesting point to study is precisely after one hour when the number of connections starts plunging down to the minimum of 3 connections. This point indicates roughly the moment the ODDSE server runs out of CPU and/or Datastore API quotas. Nevertheless the operation continues to proceed for approximately another hour. Looking at the gradient of the total number of rows being uploaded (affected rows) data throughput declined however. Somewhere shortly after the second hour passed the Datastore exceeds its storage quota. Insertion of further rows is
4. Bulk loading

Figure 4.15.: This graph presents a bulk operation loading 1,500,000 records up to an App Engine server. Due to the storage quota violation on the App Engine server the operation doesn’t succeed and blocks after 680,000 records waiting for new quotas.

hence impossible. ODDSE is waiting to resume the insertion periodically attempting to bulk load the following chunk.

Regarding this evaluation ODDSE scales up to its requirements and allows the bulk insertion of an entire Datastore.

4.3.4. Randomized bulk loading

In Section 4.1.3 a randomized bulk loading approach was proposed. It is based on research done by [SCS+08] and aims avoid hot partitions when loading data into key-range based partitions on different storage nodes. Therefore chunks, initially given in key sequence, where randomized and thereafter bulk loaded to the App Engine server. Bases of this evaluation was once again a TPC-H bulk file including 150,000 records and being 23.4 MB in size. As Bigtable relations initially consist of just one partition on a single storage node an initial bulk loading operations will certainly not show any improvement. For that reason an update scenario was evaluated altering data that was already loaded into
4.3. Evaluation

Bigtable and hopefully distributed on several storage nodes.

Figure 4.16, 4.17 and 4.18 present results retrieved when evaluating the randomized approach. Details of the evaluation are presented in Table 4.6. All randomized bulk loads show a decline in latency of 10% and above. In the first two presented cases the total duration drops correspondingly by approximately 10% as well. The third randomized bulk load is delayed due to quota violations and doesn’t perform faster than the key-sorted equivalent. Its latency improvement is however close to the second case that uses only 15 connections. This seems to indicate that there won’t be any speed up found much higher than 10% for the usage of 40 connections as well.

The expected improvement was much higher than the actual observed result in this evaluation. [SCS+08] raised expectations to gain a speed-up of about 2, accordingly an improvement of 50%. The achieved improvement of 10% is far from meeting the expectations. This might be due to several reasons:

Bigtable tablets are about 100 MB to 200 MB in size. Hence the data is most probably not split in a huge number of partitions. This would most probably give a much better improvement. The little improvement however certainly indicates that data is distributed on more than one storage node. At the same time one can only guess how partitions are split and whether they are equal in size.

To conclude, randomized bulk loading offers a small performance increase. The increase is however obtained by high costs on the ODDSE client. The preparation of all chunks in advance requires many memory resources to maintain the offset and size of each data chunk and one has to consider carefully if this pays off. If one intends to use randomized bulk loading enough memory has to be allocated to the Java Virtual Machine to pretend ODDSE from failing.
4. Bulk loading

<table>
<thead>
<tr>
<th>Static set up</th>
<th>Key sorted</th>
<th>Randomized</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>15 connections &amp; 100 rows/chunk</td>
<td>duration 501s</td>
<td>449s</td>
<td>10%</td>
</tr>
<tr>
<td></td>
<td>avg. latency 3.9s</td>
<td>3.4s</td>
<td>13%</td>
</tr>
<tr>
<td>15 connections &amp; 50 rows/chunk</td>
<td>duration 519</td>
<td>457</td>
<td>12%</td>
</tr>
<tr>
<td></td>
<td>avg. latency 2.5s</td>
<td>2.0s</td>
<td>20%</td>
</tr>
<tr>
<td>40 connections &amp; 50 rows/chunk</td>
<td>duration 453s</td>
<td>450s</td>
<td>&lt; 1%</td>
</tr>
<tr>
<td></td>
<td>avg. latency 4.1s</td>
<td>3.1s</td>
<td>24%</td>
</tr>
</tbody>
</table>

Table 4.6.: Comparison of the two bulk loading approaches

![Graph showing performance gains for randomized bulk loads](image)

Figure 4.16.: Using 15 simultaneous connections and a fixed chunk size of 50 rows per chunk this graph shows performance gains for randomized bulk loads. Both, latency and total duration drop by approximately 10%.
Figure 4.17.: This graph shows the same result than Figure 4.16 using a bigger chunk size of 100 rows.
Figure 4.18.: Increasing the number of server connections to 40 this graphs shows the expected decline in latency for randomized inserts. However both operations take the same time to succeed. The randomized bulk load is slowed down due to quota violations.
5. Query processing

ODDSE offers full SQL and AmosQL query support for wrapped Bigtable sources. In order to provide powerful high level SQL and AmosQL the lower level GQL query processing in the ODDSE wrapper has to be highly efficient, scalable and last but not least reliable.

Transparent query processing of Bigtable sources is achieved using multi-directional foreign functions. This was described earlier in Section 3.3.1: The Amos II query processor evaluates SQL and AmosQL queries and produces an execution plan based on the multi-directional foreign functions. In this execution plan access to Bigtable relations is realized calling the GQL interfacing `queryBigtable` of the ODDSE wrapper.

In addition to SQL and AmosQL queries in Amos II ODDSE supports the SQL delete statement. Delete statements are available on the ODDSE GQL interface `queryBigtable` extending the capabilities of GQL (compare to Appendix A). An example is given in the Listing below:

Listing 5.1: A delete query

```sql
/* Delete query on the GQL interface */
queryBigtable("delete from Person where ssn < '880101';");
```

In the following scalable and efficient query processing done by the ODDSE wrapper is described in detail.

Even though query processing in terms of select-project-queries is not as resource intensive as previously presented bulk loading operations, ODDSE once again faces the obstacle of limited resources and quotas on App Engine servers what makes this task essentially harder. In particular the interface `queryBigtable` is required to scale as it is likely to return large data volumes.

Listing 5.2: Advanced core cluster function `#Person`

```sql
/* Improved core cluster function for Person with costs */
```
5. Query processing

```sql
create function #Person() ->
Bag of <Charstring ssn key, Charstring name> as multi-directional
("ff" select ssn, name where {ssn, name} =
    queryBigtable("select * from Person",{}))
cost{10000,10000})
("bf" select name where {ssn, name} =
    queryBigtableLite("select * from Person where ssn=%s",{ssn})
cost{10,1})
("fb" select ssn where {ssn, name} =
    queryBigtableLite("select * from Person where name=%s",{name})
cost{20,4});
```

The above Listing 5.2 presents an advanced multi-directional foreign function implementation to define a view on the Bigtable relation `Person`. The simple cost heuristic is utilized by the Amos II query processor when creating the execution plan. Even more interesting is the optimized interface `queryBigtableLite` for all TBR predicates having bound variables. Due to equality filters their result set is assumed to be much smaller than the entire relation. This assumption is expressed by the cost heuristic `(cost, fanout)` given in Listing 5.2 as well. For the key column `ssn` of `Person` the precise maximum result size is known and obviously has the `fanout` one.

ODDSE attempts to load such result sets from Bigtable in one single request to avoid the overhead of parallel query processing. If, however, this request fails the operation has to be resumed with the most general TBR predicate. This is done by the ODDSE wrapper transparently from the Amos II query processor. Resuming query processing that way is required as resumption relies on the relation’s key column and the corresponding sort order. Due to the lack of indexes on the App Engine there is no composite index available such as `(name, ssn)`. This index however is essential to support resumption of a property (here `name`) if uniqueness is not ensured.

Imagine having several thousand persons named `John`. The call `queryBigtableLite("select * from Person where name='John'")` fails as there is no possibility to resume the query from the last row after receiving a maximum result set of 1000 Johns from the App Engine. Adding a filter `name\_id='John'` obviously doesn’t help, and the solution, an index on `(name, ssn)`, is not available. In such cases ODDSE internally falls back to a more general target binding `queryBigtable("select * from Person")`. The benefit, if `queryBigtableLite` succeeds, is significant. If it fails however, the additional time is incidental compared to the overall
For large data volumes efficient chunking of queries is crucial to improve performance with the insertion of parallelized query processing. Parallelism is achieved by the *query scheduler* that manages a pool of threads each having one server-connection assigned. *Chunk queries* provided by the *query partitioner* are scheduled and run by this scheduler as earlier explained in Section 3.3.3.

To give good results optimization of both, the chunk size as well as the number of applied server-connections has to be constantly adapted. This adaption is realized by a particular *runtime optimizer*. *query partitioner* and *runtime optimizer* are centric parts of the ODDSE query processor. For that reason both are thoroughly explained in the following sections.

The Datastore API available on the App Engine offers two ways of querying Bigtable relations. In the following two listings are presented to illustrate the different approaches to access the API. In order to give a better understanding the relation *Person* defined in Section 3.3.1 is shown here again:

Listing 5.3: A Datastore model class in Python (see Listing 3.14)

```python
# Definition of the App Engine type Person that corresponds to the relation Person (first_name, last_name, age) wrapped in ODDSE
class Person(db.Model):
    first_name = db.StringProperty(required=True)
    last_name = db.StringProperty(required=True)
    age = db.IntegerProperty()
```

Listing 5.4: Query object creation using methods (Python)

```python
# query Person using db.Query
query = db.Query(Person)
query.filter('last_name >','=','M')
query.filter('last_name <','=','N')
query.order('last_name')
query.order('first_name')
result = query.fetch(limit=10)
```

This first query approach is based on a *query object* that provides methods to define multiple filters as well as a (multi column) sort order. The method *fetch* can take a upper
5. Query processing

limit of records as well as an offset from where to start. The maximum result set returned on the API is 1000 entities. Important to notice is that an offset doesn’t reduce the result set received from Bigtable, but is applied programmatically afterwards.

Listing 5.5: Query object creation using GQL (Python)

```python
# query Person using GQL
query = db.GqlQuery(
    "SELECT * FROM Person
    WHERE last_name>='M'
    AND last_name<='N'
    ORDER BY last_name, first_name
    LIMIT 10"
)
result = query.fetch()
```

The second query approach from the listing above provides a GQL query interface. The GQL query string is parsed before calling the API. Similar limitations to LIMIT and OFFSET apply here.

The ODDSE server was initially build up on the GQL query interface. This way queries could easily be forwarded to the App Engine not requiring any parsing in the client. In order to improve performance, avoiding expensive server-side parsing of GQL strings, ODDSE uses the approach presented in Listing 5.4. The usage of `db.Query` query objects has besides that another huge advantage due to the way chunk processing is optimized in ODDSE.

In order to avoid incessant query parsing, respectively the creation of `db.Query` objects for every single chunk a caching mechanism for query objects was introduced: Whenever receiving a new query a `db.Query` is created and stored in the App Engine’s Memcache. Following chunks do not resend the query itself, but rather profit from the cached query object identified by their `queryId`. The chunk queries just have to provide additional filters according to the given sort order to adequately narrow the cached query down using `query.filter()`.

Using the query approach presented in Listing 5.4 implies parsing of GQL queries in the ODDSE client. The ODDSE wrapper transforms GQL strings in an intermediate representation based on POST parameter being efficient for usage on the ODDSE server. Nevertheless this intermediate format covers all capabilities of GQL. It is furthermore described and illustrated in Section 3.3.2, in particular in combination with the mentioned
caching mechanism.

5.1. Chunk generation

5.1.1. Sequential approach

Sequential chunks certainly refer to the easiest way to partition a queried result set. This obviously doesn’t scale as the client-server interaction is limited to a single connection. Nevertheless there are situations described later that require a fall-back to the sequential approach.

Additionally the sequential query performance was used as point of reference in order to assess later optimizations.

Sequential chunking was first done following a trial & error approach. The result set size is usually not known in advance following such a naive approach. Query processing on the App Engine server was continued until running into timeout limits. Whenever failing due to timeout failures the prepared result set is returned back to the client in combination with failure information. By means of the resumable query managers the ODDSE client is able to resume processing. Based on the partially returned result set the initial query is narrowed down. This is done rewriting the query with an additional filter to exclude all records that where already successfully received.

Even though this behavior is adaptive to the servers current work load in terms of the returned partial result set, deliberatively running into failures most probably doesn’t help to scale. Long running requests furthermore affect the App Engine quotas badly and are not advisable.

Therefore a client side managed chunk size was introduced. This chunk size is adapted by the client’s runtime optimizer. Instead of retrieving as many records as possible a fixed number of records is retrieved and - if more data is available - resumed from there.

The obstacle to performance is obviously the dependency of each chunk on its predecessor. The following approach effectively removes this dependency without any pre-knowledge and allows chunks to be processed in parallel.
5. Query processing

Figure 5.1.: This figure repeats 3.14 illustrating the iterative retrieval of chunk boundaries using a chunk cursor by means of a chunk cursor manager. Chunk boundaries are made available from a chunk queue helping to speed up query processing on the query scheduler by querying multiple data chunks simultaneously.

5.1.2. Chunk Cursor

The chunk cursor manager is a chunk iterator that offers a convenient way to parallelize query processing without any pre-knowledge for chunks generation. The idea is to iterate over a result set on Bigtable using a kind of chunk cursor. Each iteration step of this chunk cursor is initiated by the ODDSE client and returns the next appropriate chunk boundary based on a chunk size determined by the runtime optimizer. The mechanism is illustrated in Figure 3.14.

In that manner chunks are sequentially retrieved from a remote result set. An interesting question to ask is how this actually helps to parallelize query processing. The answer lies in the low latency for such chunk cursor queries taking considerably less time than retrieving the entire chunk’s data from Bigtable. While retrieving one chunk of data from Bigtable hence several chunk request of the chunk cursor manager can be run sequentially. These requests store their result (chunk boundaries) in a chunk queue as presented in Figure 3.14.
5.1. Chunk generation

With a growing size of this chunk queue more and more connections to the ODDSE server can be utilized constantly increasing parallelism in the client up to a certain level. The growing number of simultaneous chunk queries significantly increases data throughput.

The availability of chunk boundaries showed to be important to be able to run chunk queries whenever a server-connection is freed and becomes available. For this reason chunk cursor requests are run with a higher priority by the client query scheduler.

Similar to the file partitioner presented in Section 4.1.2 the chunk queue is limited to the number of server connections and blocks if further chunks are added. On the one hand reasons are similar to the ones presented in 4.1.2. Preparation of all chunks at early stage is everything but optimal as the chunk size is constantly adapted by the runtime optimizer. Every prepared chunk is however fixed in size and might either be not efficient or fail due to timeouts considering the current status of the App Engine server. On the other hand one has to keep in mind that chunk cursor requests are actually queries that impact the clients and servers performance. To minimize this impact such queries should be equally distributed over the total time processing proceeds.

As shown later in Section 5.4 chunk cursor based query chunking and simultaneous query processing performs great compared to a naive sequential approach. However do the additionally required chunk requests by the chunk cursor manager degrade the overall query performance, in particular in the beginning when no chunk boundaries are available yet.

In order to further improve performance more sophisticated approaches are required. These approaches all rely on the collection of statistical data describing data ranges and distribution of data stored in wrapped Bigtable sources. Based on such data chunks can be created locally on the client avoiding the high start up costs of the chunk cursor manager as well as the large overhead due to one additional chunk request per each query chunk execution.

5.1.3. Wavelet based histograms

ODDSE offers a more sophisticated way to chunk queries using a histogram based on Haar wavelets. The ODDSE wrapper therefore implements a histogram builder and an equivalent partitioner in the statistics package.
5. Query processing

[KH01] describes wavelets and their applications in databases. In addition to clustering techniques and approximate query processing wavelets can be utilized for selectivity estimation using wavelet-based histograms [IP95].

Selectivity estimation allows to approximate the fraction of records in a data source that satisfy a given query. In ODDSE the challenge is contrariwise to build chunk queries of a target size. Due to the runtime optimization done in ODDSE the target chunk size is moreover constantly changing. This makes chunk generation even more challenging and manifests selectivity estimation to be an essential step when attempting to narrow queries down to chunk queries.

[MVW00, MVW98] evaluate the usage of wavelet-based histograms for selectivity estimation and show their superior performance compared to random sampling and other estimation approaches. Based on the technique proposed by [MVW98] a wavelet-based histogram for ODDSE was implemented in this work.

One major advantage of wavelet-based histograms are their multi-resolution properties due to the hierarchical decomposition of wavelets [KH01]. In ODDSE these properties are substantial in order to create chunks of various sizes as shown in Figure 5.3 and Figure 5.4. Such histograms hence perfectly support the increased demands given by runtime optimization of chunk queries.

Recall the multi-directional functions of Amos II that provide the queryable view on Bigtable relations. To enhance performance in ODDSE a histogram is only required for the (composite) key of a relation to support the most general access path with no variables bound. All TBR predicates having one or more variables bound would require a second level sort order on the bound variable first, and the (composite) key second. This however is not supported by the App Engine Datastore.

The wavelet decomposition itself is loss free. However efficient compression is possible making wavelet-based histograms very space efficient.

In the following the wavelet decomposition for Haar wavelets is described to give a better understanding of the later description of ODDSE’s wavelet-based histogram and the selectivity estimation achieved. The example presented is taken from [MVW00] and adapted to the purpose of ODDSE. Haar wavelets are fast to compute and rather simple to implement [MVW00]. As they perfectly serve the purpose of efficient histograms this type of wavelets was used in ODDSE.
The Haar wavelet decomposition is applied to a one-dimensional signal of \( N_0 = 2^l \) items where \( l \) refers to the number of composition levels and hence determines the granularity of the histogram. The example presented in the following uses 3 decomposition levels. Assume the decomposition algorithm works on the following signal:

\[ S_0 = [2, 2, 0, 2, 3, 5, 4, 4] \]

Each decomposition step transforms the input signal \( S_i \) to a lower resolution signal of size \( N_{i+1} = \frac{N_i}{2} \) building pairwise averages, such as \( \frac{2+2}{2} = 2, \frac{0+2}{2} = 1 \), ... finally returning:

\[ S_1 = [2, 1, 4, 4] \]

By averaging the values information is obviously lost. To inversely retrieve the original signal additional detail coefficients are necessary. These coefficients are calculated by dividing the pairwise differences from the original signal \( S_i \), for instance \( \frac{2-2}{2} = 0, \frac{0-2}{2} = -1 \), ...:

\[ C_1 = [0, -1, -1, 0] \]

Repeating this decomposition step recursively on the newly calculated \( S_{i+1} \), we get the full decomposition:

<table>
<thead>
<tr>
<th>Level</th>
<th>Averages</th>
<th>Detail Coefficients</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( S_0 = [2, 2, 0, 2, 3, 5, 4, 4] )</td>
<td>( C_1 = [0, -1, -1, 0] )</td>
</tr>
<tr>
<td>1</td>
<td>( S_1 = [2, 1, 4, 4] )</td>
<td>( C_2 = [\frac{1}{2}, 0] )</td>
</tr>
<tr>
<td>2</td>
<td>( S_2 = [\frac{3}{2}, 4] )</td>
<td>( C_3 = [-\frac{5}{4}] )</td>
</tr>
<tr>
<td>3</td>
<td>( S_3 = [\frac{11}{4}] )</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.1.: Wavelet decomposition[MVW00]

The final wavelet decomposition is the overall average of \( S_0 \) represented by the single value of \( S_3 \), followed by all detail coefficients from bottom to top in Table 5.1:

\[ \hat{S} = [\frac{11}{2}, -\frac{5}{4}, \frac{1}{2}, 0, 0, -1, -1, 0] \]

An advantage of wavelet transformations is the exploitation of locality as detail coefficients often turn out to be 0 or at least very small in magnitude. Such small coefficients can be truncated causing only small errors when reconstructing the original signal. [MVW98] describes thresholding methods to keep the best \( m \) coefficients minimizing the error of approximation at the same time.
5. Query processing

Following a domain \( D = \{d_1, d_2, \ldots, d_m\} \) of an attribute \( A \) shall be the set of all possible values of \( A \) whereby \( D \) is an ordered set with \( d_1 < d_2 < \ldots < d_m \). \( V \subseteq D(A) \) is the partially ordered set of values for \( A \) that are actually present in relation \( R \). To use a general encoding of wavelet coefficients over possible domains \( D \) ODDSE uses a transformation \( \Phi : D(A) \rightarrow D'(A) \) to convert domains into a partially ordered set \( D' = \{d'_1, d'_2, \ldots, d'_m\} \) in the interval \([0 = \min(D'), 1 \approx \max(D')]\).

The origin of a ODDSE histogram is a equi-width histogram of very fine granularity \( (2^{-d}) \) build by retrieving all \( v \in V \) of \( A \) in their partial order. This rather primitive histogram is just used as input for a wavelet decomposition in order to build a more powerful multi-resolution histogram that facilitates generation of chunks of a desired size.

Listing 5.6 presents the ODDSE-internal encoding of wavelet coefficients for storage in Amos II. [relation, attributeId] determines a specific attribute, \( \text{lowerB} \) and \( \text{upperB} \) refer to the data interval and hence determine the level of composition. Their type \( \text{Real} \), respectively \( \text{Double} \) in Java, refers to the common domain \( D' \) from above. The overall average from \( S_l \) as described above is however not stored in ODDSE. Instead the total sum \( \sum_{s \in S_0} s \) is stored in \( \text{entity statistic} \). This sum is obviously equivalent to the number of all rows in relation \( \text{relation} \).

Storing the sum of \( S_0 \) instead of the average allows to calculate the size of partitions on all levels more easily.

Listing 5.6: Type WaveletCoefficient in ODDSE (AmosQL)

```plaintext
/* Wavelet coefficient on single property */
create type WaveletCoefficient

properties ( relation Mappedtype, attributeId Integer, lowerB Real, upperB Real, value Integer );

/* Global entity count */
create type EntityStatistic

properties ( relation Mappedtype key, count Integer );
```

In ODDSE the original signal \( S_0 = [s_{0,1}, s_{0,2}, \ldots, s_{0,2^l}] \) for wavelet decomposition represents a continuous count of attribute values \( v_k' \) within intervals of size \( 2^{-d} \) whereas \( v_k \in V \).
5.1. Chunk generation

Figure 5.2.: Wavelet decomposition in ODDSE is done on value streams in a continuous manner on newly received values and is illustrated in this figure as processing from the left to the right. Instead of decomposing level by level the histogram builder processes a level whenever the pair of coefficients on the level below is available.

corresponds to \( v'_k \in V' \) from the converted domain \( D' \):

\[
s_{0,i} = \| \{ v'_k | (i-1) \cdot 2^{-d} \leq v'_k < i \cdot 2^{-d} \} \|
\]

Decomposition of \( S_0 \) is however not done in levels as presented in the above example but rather in a continuous manner on the value stream \( v_{k-1} \leq v_k, k \in \{2, 3, \ldots, \|V\| \} \) over all levels \( l \) as illustrated in Figure 5.2. This allows a more efficient calculation of the required detail coefficients. Coefficient of value 0 can besides be immediately skipped.

Histogram generation is done by the statistics manager using a histogram builder. Currently it has to be manually initiated for a certain property available from a wrapped Bigtable source. This is done using the foreign function \( \text{buildHistogram}(\text{type}, \text{property}, \text{granularity}) \) on the Amos II user interface. Granularity thereby specifies the number of wavelet decomposition levels \( l \).

Future research should be done here on how to enable incremental updates having generated histograms once. [MVW00] describes a framework to handle incremental updates of wavelet-based histograms that could be applied in future versions of ODDSE.

The interesting question how to build chunks of specific size from a given query still remains. To explain this we’ll go back to the earlier example given to illustrate the wavelet decomposition. Table 5.1 presented the final decomposition. Remember however that ODDSE uses the overall sum \( \sum_{s \in S_0} s = 22 \) instead of the average as calculated in \( S_3 \).
5. Query processing

Figure 5.3.: This figure presents the decode tree of the wavelet decomposition done in Table 5.1 for the chunk condition $3 \leq \|P'_p\| \leq 5$. All nodes refer to chunks of the given size showing the multi-resolution properties of the wavelet decomposition. Leafs might be combined over several sub-trees such as the leafs of size 2 and 3.

Suppose we are aiming at building partitions $P'_p$ satisfying a chunk condition $3 \leq \|P'_p\| \leq 5$. Figure 5.3 illustrates this example and should help while reading. (1) Our starting point is $\sum_{s \in S_0} s = 22$ and obviously violates the condition. To get a smaller partition we descend on level to the left as illustrated. A descend refers to a division by 2 after adding (left descend) or subtracting (right descend) the detail coefficient. Due to the summed global value we furthermore have to scale the coefficients by the current level. Descending to the left thus results in $\frac{22+5/4 \times 2^3}{2} = 6 \not\geq 5$. Consequently another descend to the left is required resulting in $\frac{6+1/2 \times 2^2}{2} = 4 \leq 5$. $\|P'_1 = s_{1,1}\| = 4$ satisfies the above condition, a first partition is found. (2) Suppose we kept the last levels on a stack we are able to move up easily without any re-calculation, remembering however to descend right next. This gives a partition of size $\frac{6-1/2 \times 2^2}{2} = 2 \not\geq 3$ violating the enforced lower size. We keep the partial result $P'_2 = s_{1,2}$ however and ascend until finding the next location from where to descend right again. Descending right from the top we get $\|P'_2\| + \frac{22-5/4 \times 2^3}{2} = 2 + 16 \not\geq 5$, descending left $\|P'_2\| + \frac{16+0 \times 2^2}{2} = 2 + 8 \not\geq 5$ and finally left again resulting in $\|P'_2\| + \frac{8+1/2 \times 2^1}{2} = 2 + 3 \leq 5$. $P'_2 = s_{1,2} \cup s_{0,5}$ satisfies the chunk condition. (3) The next descend to the right after ascending gives $\frac{8-1 \times 2^1}{2} = 5 \leq 5$, partition $P'_3 = s_{0,6}$. (4) Correspondingly partition $P'_4 = s_{0,7}$ and (5) $P'_5 = s_{0,8}$ are calculated.

Figure 5.4 illustrates another example using the same wavelet coefficients, but a different chunk condition $5 \leq \|P'_p\| \leq 10$. The multi-resolution properties of wavelets are evidently
Figure 5.4.: This figure presents the decode tree of the wavelet decomposition done in Table 5.1 for the chunk condition $5 \leq \|P'_p\| \leq 10$. All nodes refer to chunks of the given size showing the multi-resolution properties of the wavelet decomposition.

well suited for efficient generation of partitions of various granularities.

The retrieved partitions are however still represented in the internal domain $D'$ whereby $\forall v' \in V' \subseteq D'(A) : v' \in [0, 1)$. In order to apply the retrieved partitions to the initial input query a inverse transformation $\Phi^{-1} : D'(A) \rightarrow D(A)$ has to be done. While $D(A)$ was a totally ordered set $D'(A)$ is only partially ordered. The total order in $D(A)$ is thus lost after a transformation back. However, a partial order is sufficient and can be ensured:

$$\Phi(a) = \Phi(b) \iff \Phi^{-1}(\Phi(a)) = \Phi^{-1}(\Phi(b)) \leq \min(a, b) \text{ whereby } a, b \in D(A)$$

5.2. Runtime optimization

Runtime optimization in terms of bulk loading was already thoroughly explained in Section 4.2. For several reasons presented in that section optimization of bulk loads was much more difficult and therefore subject to the main work done in optimization.

The runtime optimizer for query processing shares the main ideas with the optimizer presented earlier. This section hence focuses above all on distinct requirements and emphasizes the differences between both optimizers. Nevertheless there are some important similarities to be mentioned, of course.

Once again optimization of the client-server interaction is crucial to ensure high performance and scalability. Reliability by contrast showed not to be as critical as for bulk insertions. This is certainly due to the fact that expensive write operations for entities in a distributed environment show a lot more point of failures than simply querying entities.
5. Query processing

At the same time the impact of App Engine failures on the ODDSE’s performance is way smaller. If a query fails due to a timeout on the App Engine it anyhow returns the entities that were processed up to then. For timeouts on the Datastore API however nothing except the failure is returned. But due to cheaper operations the frequency of timeouts during query processing is considerably smaller than the one observed for bulk insertions.

A failure of bulk chunks requires reloading this data to the App Engine server. Due to network latency these reloads are very expensive, in particular when done more than once. Failed query chunks on the other side don’t require any repeated uploads. Lots more there is actually almost no difference whether queries are split on the client by the query partitioner or on the server due to timeouts. The only difference remains in resource consumption on the App Engine. Due to Google’s policy affect long running queries your quotas more than average ones.

This however doesn’t say chunks are unnecessary for query processing. They remain essential to utilize multiple simultaneous connections to the App Engine and hence significantly improve data throughput. The usage of reasonable sized chunks additionally allows to balance the work load on the client among available worker threads and server-connections. As soon as a chunk is processed in the query scheduler it is locked to a certain worker thread using one single connection to the App Engine. If, however, such a chunk is rather huge compared to others it might outlast the processing of all the others. The processing of the remaining data is locked to one single connection and ends up being sequential.

The above analyzes lead to some distinct requirements for the query runtime optimizer:

- As query processing is much cheaper such chunks should be considerably bigger than bulk loading chunks. This however implies that the average of succeeded chunks used by the runtime optimizer has to be maintained separately for both types.

- Failures are cheap compared to bulk loading failures. Additional resource allocation can hence be done more often and more insisting following a greedy approach.

- As the risk of missing reliability is much smaller less resources have to be freed in case of failures.

Similar to the bulk loading optimizer optimization in terms of query processing is done in two opponent steps. All formulas used were explained in Section 4.2 and are therefore mentioned without further notice.
Resource allocation steps are done before executing query chunks on the query scheduler and generally increase the size of chunks regarding timeouts and the number of server connections regarding quota failures if:

\[ t_{\text{now}} - \max(t_{\text{allocation}}(\text{failure}), t_{\text{recovery}}(\text{failure})) > I_{\text{allocation}}(\text{failure}) \]

whereby:

\[ I_{\text{allocation}}(\text{failure}) = I_{\text{defaultAllocation}} \times \text{density}_2(\text{failure}) \]

\[ I_{\text{defaultAllocation}} \] in case of query processing is lower than \( I_{\text{defaultAllocation}} \) applied for bulk loading to allocate resources more frequently.

Failure decrease steps are done in response to server processing failures and generally decrease the chunk size regarding timeouts and reduce the number of utilized connections regarding quota failures if:

\[ t_{\text{now}} - t_{\text{recovery}}(\text{failure}) > I_{\text{defaultRecovery}} \]

In case of \( I_{\text{defaultRecovery}} \) always the same value is applied. This however results in less frequent failure decrease steps for query processing optimization as query chunks generally show lower latency.

During both steps parameters are adapted using the earlier defined sigmoid function \( \text{adapt}(x, \text{failure}) \):

\[ \text{adapt}(x, \text{failure}) := \frac{f_{\max} + f_{\min}}{2} + \frac{f_{\max} - f_{\min}}{2} \times \frac{h(x, \text{failure})}{\sqrt{1 + h(x, \text{failure})^2}} \]

whereby:

\[ h(x, \text{failure}) := \frac{s}{\text{density}_{10}(\text{failure})} \times (x_{\text{opt}} - x - \frac{1}{s} - \log_2(x_{\text{opt}}) \times \text{density}_2(\text{failure})) \]

Relevant parameters for runtime optimization of queries in the above opponent steps are the following:

Simultaneous connections (query scheduler) The consideration done for the number of simultaneous server-connections used for query processing is similar to the one presented in case of bulk loading. In similar manner Figure 5.6 and Table 5.2 support the assumptions and the chosen strategy.

\(^1\) Consult Section 4.2 for a detailed description of \( \text{adapt}(x, \text{failure}) \).
5. Query processing

The number of simultaneous connections \textit{conns} is adapted by means of the sigmoid function \textit{adapt}(x, failure). Incremental and declining adaption use both their own parameter set:

\[
\text{adapt}_{\text{dec}}(\text{conns}, \text{failure}) : \text{conns}_{\text{opt}} = 22; f_{\text{min}} = 1.01; f_{\text{max}} = 1.15; s = \frac{1}{4}
\]

\[
\text{adapt}_{\text{inc}}(\text{conns}, \text{failure}) : \text{conns}_{\text{opt}} = 22; f_{\text{min}} = 1.005; f_{\text{max}} = 1.10; s = \frac{1}{4}
\]

The target number of 25 connections as \( x_{\text{opt}} \) is a primitive heuristic showing results.

**App Engine chunks (query partitioner)** The chunk size is again adapted by means of the sigmoid function \textit{adapt}(x, failure). Incremental and declining adaption use both their own parameter set:

\[
\text{adapt}_{\text{dec}}(\text{rows}, \text{failure}) : \text{rows}_{\text{opt}} = \text{rows}_{\text{avg}}(t_{\text{now}}); f_{\text{min}} = 1.005; f_{\text{max}} = 1.15; s = \frac{1}{20}
\]

\[
\text{adapt}_{\text{inc}}(\text{rows}, \text{failure}) : \text{rows}_{\text{opt}} = \text{rows}_{\text{avg}}(t_{\text{now}}); f_{\text{min}} = 1.005; f_{\text{max}} = 1.10; s = \frac{1}{20}
\]

whereby:

\[
\text{rows}_{\text{avg}}(t) := \frac{\sum_{i=1}^{5000} \text{rows}(t - i)}{5000}
\]

The target size \( \text{rows}_{\text{opt}} \) refers to a floating average of the last thousands successfully inserted rows per chunk.

**Block time (query scheduler)** The adaptive block time is managed as part of the shared recovery data that was described in Section 3.3.3 and is calculated similar to \( I_{\text{allocation}}(\text{failure}) \):

\[
I_{\text{block}}(\text{QuotaFailure}) := I_{\text{defaultBlock}} \times \text{density}_e(\text{QuotaFailure})
\]

Such as \( I_{\text{defaultAllocation}} \) is \( I_{\text{defaultBlock}} \) for similar reasons smaller in case of query processing.

5.3. Inheritance

ODDSE fosters the data model of Amos II and even supports inheritance. This is possible due to the App Engine Modeling API. Instead of relations the data model is actually based on entity types in terms of Python classes which can be subclassed (see 2.3.1).

Nevertheless multiple inheritance, as supported by Amos II [RJK04], is not supported.
A design decision applying inheritance has to be made at an early stage. Later schema adaptations changing an existing type hierarchy will only affect newly created records. Bigtable itself doesn’t support inheritance natively. Nevertheless, the App Engine Modeling API provides a *PolyModel* providing inheritance support. Under the covers of this polymorphic model all records are stored in Bigtable in the same root relation keeping a hidden meta attribute that specifying a record’s type [Goo08].

Recall Listing 3.3 creating relation *Person*:

```
Listing 5.7: Creation of a new Bigtable relation (Listing 3.3)
createBigtable(
    "Person", // relation name
    {"ssn","name"}, // attributes
    {"Charstring","Charstring"}, // attribute types
    {"ssn"} // key attributes
);
```

```
Listing 5.8: Creation of a new Bigtable relation with inheritance
createBigtable(
    "Customer", // relation name
    "Person", // parent relation
    {"income"}, // attributes
    {"Integer"}, // attribute types
);
```

The *core cluster function* created for Customer provides a view *Customer(ssn, name, income)*. Querying the view *Person(ssn, name)* will return Customers as well.

The inheritance model provided is however very restricted. It is not possible to add additional key attributes for subtypes. Otherwise uniqueness of keys of supertypes couldn’t be ensured.
5. Query processing

Figure 5.5.: This figure illustrates query processing in ODDSE in terms of four different query and chunk generation strategies. Naive sequential processing, using one connection only, is significantly slower than parallelized approaches which utilize multiple connections to Bigtable simultaneously. The chunk cursor retrieves chunk queries directly from Bigtable. The wavelet-based partitioner builds chunk queries from local wavelet-based histograms.

5.4. Evaluation

Basis for this performance evaluation is the TPC-H benchmark data, that was uploaded in the previous section. Similar to bulk loading operations the performance of query processing in ODDSE depends significantly on multiple simultaneous server-connections. Table 5.2 evaluates the four different query strategies that were presented in Figure 5.5.

Similar to bulk uploads a huge speed up is achieved applying several server-connections simultaneously. Table 5.2 evaluates the four different query strategies that were presented in Figure 5.5. The total number of rows returned was $\approx 65,000$, respectively 10 MB. Between the naive sequential strategy and a parallel query strategy based on the wavelet partitioner a speedup of $> 8$ was achieved.
5.4. Evaluation

<table>
<thead>
<tr>
<th>ODDSE set up</th>
<th>Duration</th>
<th>Rows</th>
<th>Throughput</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>static chunks, sequential</td>
<td>407s</td>
<td>64,373</td>
<td>25.2 kB/s</td>
<td>succeeded</td>
</tr>
<tr>
<td>adaptive chunks, sequential</td>
<td>271s</td>
<td>64,373</td>
<td>37.8 kB/s</td>
<td>succeeded</td>
</tr>
<tr>
<td>chunk cursor</td>
<td>110.2s</td>
<td>64,373</td>
<td>92.9 kB/s</td>
<td>succeeded</td>
</tr>
<tr>
<td>wavelet partitioner</td>
<td>48.8s</td>
<td>64,373</td>
<td>209.8 kB/s</td>
<td>succeeded</td>
</tr>
</tbody>
</table>

Table 5.2.: Evaluation of Figure 5.5 (ODDSE query strategies)

Figure 5.6 evaluates the latency of the four queries presented in Figure 5.5. The latency of the chunk cursor is slightly misleading in this chart as an averaged latency is presented. Every second request using a chunk cursor based parallel query strategy is a returns just a single result, the next chunk bound. Such requests have a very low latency making it possible to realize parallelism with chunks directly retrieved from Bigtable. The averaged latency is therefore even below the one measured for sequential query strategies.

The latency of the adaptive sequential approach is constantly increasing due to increasing chunk sizes. The latency of the wavelet-partitioner based query strategy is exceptionally high in the beginning, as visible from Figure 5.10 as well. The reason for this peak is the Python query object (recall Listing 3.18), which is shared among all chunks by the App Engine memcache. After creation the query object might not be immediately available from the caching system, however. In particular the wavelet-partitioner based query strategy is able to utilize all available connections from the very first moment. If the shared query object is then delayed, latency increases significantly.

Figure 5.7 presents again the very good results achieved using parallelized query strategies based on chunk queries. Instant chunk queries build on statistical data by a wavelet histogram based partitioner are obviously superior to chunks build by a chunk cursor directly on Bigtable. Nevertheless the latter is important to have as it doesn’t require any analyzes beforehand. Chunk cursors hence provide an efficient tool to collect statistical data and to serve as data basis when building the wavelet-based histograms.

When evaluating the performance of chunk cursor based queries, the data throughput is interesting to observe. The throughput for wavelet-partitioner and chunk cursor based queries is presented in Figure 5.8. While the first shows a rather constant throughput, throughput for the latter is increasing until reaching a certain level. The reason is simple: Chunk cursor based query processing starts sequentially as no chunk query is available. As
Figure 5.6.: This figure presents the latency measured for several queries. The \textit{wavelet-based partitioner} provides \textit{chunk queries} instantly, allowing a high parallelism right from the start, hence showing the highest latency similar to the observations made for bulk loading operations.

Soon as more and more chunk cursor requests succeed due to their low latency, parallelism for query processing is constantly increased according to the number of new chunk queries available. As soon as all server-connections are utilized data throughput has reached its upper limit.

Figure 5.9 evaluates different parameter sets for queries utilizing the wavelet-based partitioner. The figure clearly indicates an upper limit of data throughput. Trying to utilize more resources in terms of bigger chunks and more server-connections doesn’t perform well, but rather imposes a negative impact and slows processing down. It is difficult to estimate the reasons for this. Most probably client processing is slowed down due to an increasing overhead of task switches on the single core machine, but having an increasing work load on the other hand.

The \textit{runtime optimizer} adapts the number of server-connections towards this gate, observed around 25 simultaneous connections.
Figure 5.7.: This figure compares chunk generation based on the chunk cursor and the wavelet-based partitioner. The chunk cursor obviously returns constantly growing chunk queries as defined by the runtime optimizer. The wavelet-based partitioner uses a lower and upper gate when building chunks and is therefore less sensitive to chunk size adaptations. Its sensitivity depends on the granularity of the wavelet-based histogram that serves as data bases for the partitioner.
5. Query processing

Figure 5.8.: Once again chunk generation based on the chunk cursor and the wavelet-based partitioner is compared in this figure. The query based on the wavelet partitioner shows a rather constant throughput as chunk queries are instantly available. The throughput achieved using chunk queries provided by a chunk cursor is very low at the beginning, but increasing as soon as chunks are loaded from Bigtable allowing to utilize more and more connections. Due to the large overhead (additional chunk requests for instance) the performance of the wavelet based operation remains superior.
Figure 5.9.: This figure evaluates three queries based on a wavelet partitioner. It indicated an upper limit for the optimal number of connections around 25 such as modeled in the adaption function of the runtime optimizer. Additional connections utilized by the later two operations don’t give any performance benefit, but rather take slightly more time.
Figure 5.10.: The figure analyzes the latency of the queries presented in Figure 5.9. The last query utilizing most server-connections shows, as expected, generally the highest latency. Interesting to observe is the peak at the beginning of all queries. This is most probably due to the cached Python query object on the ODDSE server (recall Listing 3.18). The query object is created by the first chunk query and might not instantly be available to all other chunk queries from the caching system.
6. Related work

Several other projects working with the Google App Engine show similar objectives than ODDSE. All of these projects address the presented lock-in issue and therewith certainly stress its importance.

**jiql [JIQ09a]** is a JDBC wrapper for accessing the Google DataStore using standard SQL.

jiql however is in a early stage of development and there is hardly any information available on jiql’s work manner. According to [JIQ09b] jiql is capable of processing JOINS such as ODDSE. In contrast to ODDSE jiql is not able to deal with large datasets as timeout failures are not handled by the system [JIQ09b]. jiql furthermore requires the definition of filters in the SQL *WHERE* statement according to their selectivity lacking an appropriate query optimizer.

**Google’s bulk loading tool [Goo08]** This tool was already earlier discussed when evaluating the ODDSE bulk loading approach and was shown to scale less good than ODDSE.

**AppRocket [App09]** follows a different approach and replicates data stored in Bigtable to a local (in client) MySQL database and vice versa. All data manipulation and query processing can be done locally outside the tight request limits of the App Engine. Moving large data volumes back and forth might be still challenging as no focus was put on fault-tolerance. Especially replicating larger datasets from the Google App Engine will be essentially slowed down due to the usage of timestamps specifying the last replication done.

**Gaebar (GAE Backup and Restore) [Bal08]** is entirely webbased and focuses on backup and restore operations only. Small chunks of data are encoded in Python code fragments and downloaded on the locale development server. If a restore operation is required the Python chunk files are uploaded to the App Engine server. Running the code fragments the original data is written back into Bigtable.
6. Related work

The purpose of this project is obviously not to provide more powerful query facilities. Nevertheless there are some severe drawbacks choosing Gaebär’s design. Encoding data in Python code implies a large overhead consuming both, CPU cycles and data volume. The python code fragments are furthermore stored in Bigtable and asynchronously downloaded from there by a locale client. When storing large datasets this might be impossible due to storage limitations. Based on the experiences made with ODDSE when dealing with large data volumes such an approach will hence hardly scale.

[SCS⁺08] discusses an approach for efficient bulk loading of records into distributed ordered tables which are range-partitioned according to their key over a large cluster of shared nothing machines. Such distributed storage systems are for example Yahoo PNUTS [CRS⁺08] or Google’s Bigtable [CDG⁺06] which is wrapped by ODDSE in this project. [SCS⁺08] proposes a bulk insertion framework based on three phases: staging of data, planning for the bulk insertion and finally the actual insertion operations. The proposed framework however relies on internal specifics such as partition sizes, the number of partitions, the way partitions are split and redistributed and several more. Bigtable however is a black box for ODDSE and such specifics are not known. ODDSE therefore follows another approach mentioned by [SCS⁺08]. Randomizing key ordered data supports the Bigtable to achieve a better load balancing.

[IP95, MVW00, MVW98] study the usage of histograms for selectivity estimation. The evaluation done by [MVW98] shows improved performance of histograms using wavelet decomposition compared with random sampling and other approaches. Based on the proposed method for building efficient and effective histograms by [MVW98] a simple, but sufficient wavelet-based histogram for ODDSE was implemented in this work.
7. Conclusions and future work

The main contribution of ODDSE is a transparent, scalable and reliable access to Bigtable databases on various App Engines.

By leveraging the Amos II query processor to compensate for missing query facilities in GQL ODDSE enables general queries in SQL and AmosQL on Bigtable relations even allowing combination with heterogeneous data from various other sources. Being transparent to the user one doesn’t have to struggle with Bigtable’s limitations anymore but just use familiar powerful SQL.

In addition to the provision of query interfaces ODDSE provides an interface for bulk uploads and mass insertion of data into Bigtable relations. Having both, query facilities enabling local backups as well as bulk uploads, ODDSE is even convenient for restoring Bigtable relations in case of failure.

A fault-tolerant reliable and adaptive management of interaction with App Engine servers as implemented by ODDSE is fundamental to provide query and bulk loading interfaces such as above. In particular the use of parallelism showed to be crucial. Continuously optimizing parallelism ODDSE provides high performance in terms of data throughput and is hence able to deal with huge data volumes.

Having other cloud storage systems such as HBase build according to Bigtable’s design notes in [CDG+06] ODDSE can even be further utilized.

At a higher level ODDSE evaluates efficient utilization of clouds from a client’s perspective. It leverages the huge aggregated computational power of clouds in contrast to generally low resources of single virtualized application servers. As ODDSE’s interaction management is rather general it should be applicable for other kind of cloud interactions providing reliability and performance whenever being restricted by limited resources on virtualized machines.

Nevertheless there are limitations of ODDSE. Currently ODDSE doesn’t support UP-
DATEs or INSERTs on the SQL or AmosQL query interface. The interfaces are restricted to SELECT and DELETE queries. Supporting UPDATE and INSERT statements on the query interface requires further work. Nevertheless as ODDSE offers a powerful tool for bulk upload of data there are ways to get around this limitation.

Further work has to be done also on query optimization. Currently the translation of SQL or AmosQL into GQL is not always efficient. For example a SQL query in ODDSE using one single inequality filter will download the entire relation. Whenever this filter returns just a small part of the original relation this strategy is obviously not optimal. In such cases a GQL query stated on ODDSE’s internal GQL interface can be more efficient. However the GQL interface doesn’t provide the same expressiveness due to the GQL restrictions. This can be solved by using rewrite techniques to generate SQL strings [Han07].

Another restriction applies to authentication and user management. Currently ODDSE doesn’t implement such functionality. Both functionalities would certainly help to increase the safety of a running system. This point hence has to be considered carefully when setting up a public ODDSE server. Nevertheless there are easy ways to achieve a basic level of security even at the current level. They are described in Appendix B as part of ODDSE’s installation instructions. The later explained mechanisms however follow more or less the idea of Security by Obscurity, a certainly controversial and generally not advisable methodology. The implementation of an authentication and user management system exceeds the scope of the current report and remains future work as well.

ODDSE was evaluated using the HTTP protocol. HTTPS however it the preferred choice when exchanging critical data between Amos II and App Engine servers. As ODDSE dynamical adapts interaction according to the App Engine’s workload increased demands put by encryption should not affect ODDSE badly. Performance in presence of encryption has to be evaluated closely though in the future.

ODDSE does not provide full ACID\(^1\) consistency. This sacrifice follows Brewer’s conjecture of the feasibility of consistent, available and fault tolerant web services [GL02]. According to Brewer it is impossible for a web service such as the ODDSE client to provide the following three guarantees at the same time:

- Consistency
- Availability

\(^1\) Atomic, Consistent, Isolated, Durable
• Fault tolerance

In order to scale large and to offer high reliability and fault-tolerance ODDSE clearly focuses on the later two guarantees. Hence ODDSE doesn’t provide ACID transactions. [BFG+08] describes how to add additional weak guarantees on top of Amazon’s storage service S3. The proposed mechanisms could be applied on top of the App Engine’s Datastore to a large extent as well. Exploring the usefulness and impact of such additional guarantees should be done as part of future work.

With the publication of Protocol Buffers by Google beginning of this year new chances for ODDSE appeared. Protocol Buffers are a way of encoding structured data in an efficient format and are used for almost all internal RPC protocols at Goggle [Goo09] including communication between the App Engine and Bigtable. According to Google Protocol Buffers are about 3 to 10 times smaller and 20 to 100 times faster than a comparable XML representation of data [Goo09]. This supports the point made earlier demanding a lightweight microformat for ODDSE.

Google’s data interchange format has a high potential to increase performance of ODDSE as it dispenses any server processing and just requires forwarding of buffers to the Bigtable API. However, ODDSE already accesses several lower level, partly undocumented Datastore API features to improve performance. Relaying on an even deeper involved access implies an increasing risk. Lower levels of an API might always be subject to changes as they are generally not intended to be accessed by developers.

Listing 7.1: A Protocol Buffers definition

```protobuf
message Person {
  required string first_name = 1;
  required string last_name = 2;
  optional int8 age = 3;
}
```

The format should be analyzes comparing risks and performance benefits in a future version of the ODDSE protocol.
Bibliography


[CDG+06] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E.


[Dis08] Google Groups Discussion. Having doubts about AppEngine, August 2008. [http://groups.google.com/group/google-appengine/browse_thread/thread/674849930bd256da/6f0d77cb40224a7a](http://groups.google.com/group/google-appengine/browse_thread/thread/674849930bd256da/6f0d77cb40224a7a).


**BIBLIOGRAPHY**

<table>
<thead>
<tr>
<th>Reference</th>
<th>Description</th>
</tr>
</thead>
</table>


BIBLIOGRAPHY


List of Figures

1.1. Tends in distributed computing ......................................... 2
2.1. ERM notation according to [Che76] ................................. 7
2.2. Cloud service levels ......................................................... 12
3.1. Architecture overview of ODDSE ................................. 18
3.2. System overview: Query processing ............................... 19
3.3. System overview: Schema processing ......................... 20
3.4. System overview: Bulk uploads ......................................... 21
3.5. Splitting of queries into chunk queries ................. 23
3.6. Metadata in ODDSE ......................................................... 29
3.7. Usage of metadata ......................................................... 30
3.8. Comparison of equi-width and equi-depth histograms ........ 34
3.9. A lightweight protocol for ODDSE ............................... 36
3.10. Interaction of components in the ODDSE client ............ 39
3.11. Class diagram: Implementation of resumable query managers in ODDSE .... 40
3.12. Class diagram: Failure types modeled in ODDSE ........... 41
3.13. Class diagram: chunk generation ........................................ 42
3.14. Illustration of chunk cursors ......................................... 43
3.15. Class diagram: The query scheduler ......................... 44
3.16. Class diagram: Hierarchy of resumable query managers and runtime optimizers ........................................ 46
3.17. Class diagram: Usage of shared recovery data ........... 48
3.18. Adaption of optimizer intervals ....................................... 49
3.19. Class diagram: The importance of the execution context .......... 50
3.20. The ODDSE server ......................................................... 51
3.21. The modules of the ODDSE server ............................... 52
LIST OF FIGURES

4.1. Fixed versus adaptive parameters for bulk loading .......................... 59
4.2. Latency of sequential bulk uploads ........................................ 62
4.3. Class diagram: Value streams in the ODDSE client for bulk loading and
    querying ................................................................. 63
4.4. Latency of sequential and parallel uploads .................................. 65
4.5. Usage of MapReduce to access Bigtable tablets ............................... 66
4.6. Optimization of connections for bulk loading ............................... 71
4.7. Chunk optimization for bulk loading ......................................... 73
4.8. Timing parameters for bulk loading .......................................... 74
4.9. Impact of adaptive and static parameters ................................... 75
4.10. Parallelization and chunk size .............................................. 77
4.11. Comparison: greedy resource allocation ..................................... 78
4.12. The top-runner of Figure 4.11 ............................................. 79
4.13. Moderate optimization for bulk loading ..................................... 79
4.15. Exceeding the available storage space ..................................... 82
4.16. Randomized bulk loading: 15 connections and 50 rows per chunk ...... 84
4.17. Randomized bulk loading: 15 connections and 100 rows per chunk ...... 85
4.18. Randomized bulk loading: 40 connections and 50 rows per chunk ...... 86

5.1. Illustration of chunk cursors (Figure 3.14) ................................. 92
5.2. Wavelet decomposition on a value stream .................................... 97
5.3. Decoding of wavelet coefficients ............................................ 98
5.4. Decoding of wavelet coefficients ............................................ 99
5.5. Comparison of query strategies ............................................. 104
5.6. Latency of different query strategies ...................................... 106
5.7. Chunk cursor versus wavelet partitioner (I) ................................ 107
5.8. Chunk cursor versus wavelet partitioner (II) ............................... 108
5.9. Query processing based on wavelet partitioner ............................. 109
5.10. Query processing (latency) based on wavelet partitioner ............... 110
## List of Tables

4.1. The Google bulk loading failure ........................................ 58
4.2. Evaluation of Figure 4.1 (ODDSE bulk loader) ...................... 60
4.3. Evaluation of Figure 4.1 (Latency for sequential bulk uploads) .......... 61
4.4. Evaluation of Figure 4.9 (ODDSE bulk loader) ...................... 76
4.5. Evaluation of Figure 4.11 (ODDSE bulk loader) ................... 77
4.6. Comparison of the two bulk loading approaches ..................... 84

5.1. Wavelet decomposition [MVW00] ........................................ 95
5.2. Evaluation of Figure 5.5 (ODDSE query strategies) .................. 105
Glossary

**ACID** The ACID (Atomicity, Consistency, Isolation, Durability) properties guarantee database transactions to be processed in a reliable manner.

**AmosQL** Amos II provides the declarative functional query language AmosQL which is relationally complete.

**Bigtable** Google’s scalable distributed cloud storage system. A Bigtable is a distributed, persistent multidimensional sorted map.

**Chunk** A chunk query is a query that returns a partition of the result of a given owner query (recall Figure 3.5). A data chunk in terms of bulk uploading correspondingly refers to a partition of a given bulk data file. ODDSE uses chunks to process operations in parallel on the App Engine.

**Cloud computing** A computing cloud is a set of network enabled services, providing scalable, QoS guaranteed, normally personalized, inexpensive computing infrastructure on demand, which could be accessed in a simple and pervasive way [WVLKT08].

**Core cluster function** An Amos II core cluster function is a multi-directional function that defines a queryable view of a relation and is furthermore exposed as SQL view providing SQL queries to that relation. Core cluster functions enable Amos II to translate and rewrite SQL or AmosQL queries in order to access wrapped relations.

**Datastore** The App Engine Datastore is a simple cloud storage system build on top of Bigtable accessible by means of the Datastore API. It offers persistent scalable storage of data in Bigtable as well as limited query facilities in GQL.

**DBMS** Database management system
Glossary

**DbRelation** The Bigtable relation DbRelation represents the ODDSE server data dictionary. Instances of the data dictionary, called *DbRelations*, are dynamically translated to Datastore model classes in Python according to the Datastore API hence providing access to Bigtable.

**ERM** Entity-relationship model

**GFS** Google file system

**GQL** The *Google query language* GQL is a relationally non-complete query language which is restricted to single table SELECT-PROJECT queries. See the language reference in Appendix A.

**IaaS** Infrastructure as a Service

**Mapped type** An Amos II *mapped type* represents an object oriented view on the state of an external data source [Geb99] and can be regarded as foreign types in Amos II defined by a core cluster function.

**Memcache** *Memcache* is a highly available hash map accessible from the App Engine platform and allows efficient caching of serialized Python objects.

**Multi-directional function** *Multi-directional functions* represent different implementations for a function to compute each of its inverses. Each such implementation is called a TBR foreign predicate and is linked to a specific *binding pattern*. *Multi-directional functions* allow a larger class of executable queries and give transparent access from AmosQL to special data structures such as wrapped external data sources.

**PaaS** Platform as a Service

**Query partitioner** The query and data partitioner is a component of the ODDSE client capable of partitioning GQL and bulk upload queries into small chunks for efficient processing in parallel.

**Query scheduler** The *query scheduler* is a component of the ODDSE client running query chunks when a connection to the ODDSE server becomes available. The chunks are run using their owner *resumable query manager*. 
**Glossary**

**RDBMS** Relational database management system

**Resumable layer** This Python module of the ODDSE server enables the functioning of *resumable query managers* in the client by providing all necessary information to continue query processing after failures.

**Resumable query manager** The *resumable query manager* controls processing of queries and bulk upload operations. In order to be fault tolerant and fulfill the centric demand for reliability in ODDSE it implements a resume mechanism allowing to recover from cloud failures and to continue queries.

**Runtime optimizer** The *runtime optimizer* is a component of the ODDSE client that continuously adapts parameters of the query scheduler and the query partitioner in order to adapt to failures and to optimize the data throughput.

**SaaS** Software as a Service

**Schema manager** The *schema manager*, a component of the ODDSE client, maps Bigtable data elements to Amos II elements by transforming Bigtable relation schemas into corresponding Amos II ODDSE schema representations and vice versa.

**SQL** SQL (Structured Query Language) is the standard query language used in most DBMSs and is relationally complete.

**Statistics manager** The *statistics manager* is a component of the ODDSE client used for GQL processing only. Providing a histogram maintaining the distribution of values of single columns the statistics manager helps to split queries into small chunk queries.

**Tablet** A *tablet* represents a key range based partition of a Bigtable and is about 100 MB to 200 MB in size.

**TBR** Type binding resolved
A. GQL-Reference

SELECT * FROM <kind>
   [WHERE <condition> [AND <condition> ...]]
   [ORDER BY <property> [ASC | DESC] [, <property> [ASC | DESC] ...]]
   [LIMIT [<offset>,] <count>]
   [OFFSET <offset>]
<condition> := <property> {< | <= | > | >= | = | != } <value>
<condition> := <property> IN <list>
<condition> := ANCESTOR IS <entity or key>
B. Installation

The following instructions help to set up a fully working ODDSE system including the creation of an App Engine account. In the presented installation however ODDSE uses only one ODDSE server even though several servers could be utilized in the system.

1. Installation of Amos II

ODDSE is a wrapper for Amos II and requires this system to be installed locally. Amos II can be obtained from http://user.it.uu.se/~udbl/amos/. Setup instructions for Amos II are available at http://user.it.uu.se/~udbl/software/setup_instructions.txt

2. Compilation of ODDSE

Run compile in the ODDSE directory:

compile.cmd

3. Creation of an App Engine application

Sign up for an App Engine account at http://code.google.com/intl/sv-SE/appengine/ and create an application using the proposed identifier: oddse\username;.

4. Installation of the Google AppEngine SDK

In order to install the ODDSE server you need Google’s SDK. If you are using an already setup App Engine you can skip this step.

The SDK requires a Python environment. If Python is not available on your system you can install it from http://www.python.org/download/

Afterwards download the App Engine SDK for Python and install it:
http://code.google.com/intl/sv-SE/appengine/

5. Security consideration for production systems
Currently ODDSE provides no user management or authentication mechanisms. If you intend to use ODDSE for a production system, security fixes have to be applied in order to disallow unauthorized access.

The ODDSE server is located in

    lib

`DatastoreConnector`. The Python file `simpleProtocol.py` defines the URI of ODDSE event handlers. These URIs can be adapted by changing the value of the module variable `oddseAccessPath`. By default this variable is set to `simpleprotocol`. If changed all Bigtable URIs following have to be adapted correspondingly, of course.

Hiding the ODDSE event handlers a primitive level of security can be achieved. Feel free to evaluate (and adapt) ODDSE for usage with HTTPS in addition.

### 6. Installation of the ODDSE server

ODDSE provides an installation script that automatically uploads the server files:

```
installAppServer.cmd oddse%username% <version>
```

The current version of the ODDSE server is 13.

After uploading the current version has to be activated as default version in the administration panel of Google App Engine:

```
http://appengine.google.com/deployment?app_id=oddse<username>
```

### 7. Running ODDSE:

```
oddse.cmd
```

### 8. Creation of a Bigtable relation:

```
createBigtable( "http://oddse"+getenv("username")+
    "..appspot.com/simpleprotocol",
    "Person",
    {"ssn","name"},
    {"Charstring","Charstring"},
    {"ssn"} );
quit;
```

The new entity type `Person` can now be seen in the App Engine Data Viewer on the administration panel.
9. Population of a Bigtable relation with ODDSE:

```
oddse.cmd

accessBigtable("http://oddse"+getenv("username")+
   ".appspot.com/simpleprotocol");

bigtableUpload("dat/person.dat","Person",{"ssn","name"});
quit;
```

10. Query Bigtable relation *Person*:

```
oddse.cmd

accessBigtable("http://oddse"+getenv("username")+
   ".appspot.com/simpleprotocol");

sql("select * from Person");
sql("select * from Person where ssn = '820614-338'");
sql("select * from Person where name = 'Moritz Mack'");
sql("select * from Person where ssn >= '820614-338'
   
   and name = 'Moritz Mack'");
sql("select * from Person where ssn > '820614-338'");
sql("select * from Person p, Person q where p.name=q.name");
quit;
```
C. ODDSE reference

queryBigtable(query)

Generic GQL query interface that is optimized for large result sets. Uses simultaneous connections to the ODDSE server to maximize throughput.

This query interface allows GQL and additionally DELETE queries.

Parameters

query: SQL query statement

(type=Charstring)

Return Value

Result set

(type=Bag of Vector)
### queryBigtable(query, params)

Generic parameterized GQL query interface that is optimized for large result sets. Uses simultaneous connections to the ODDSE server to maximize throughput.

This query interface allows GQL and additionally DELETE queries.

**Parameters**

- **query**: prepared SQL query statement
  
  \( (type=\text{Charstring}) \)

- **params**: Vector of parameters for prepared statement
  
  \( (type=\text{Vector}) \)

**Return Value**

Result set

\( (type=\text{Bag of Vector}) \)

---

### queryBigtableLite(query)

Generic GQL query interface that is optimized for small result sets. Attempts to query result set using one single connection. If this fails fallback to parallel processing using multiple connections.

**Parameters**

- **query**: SQL query statement
  
  \( (type=\text{Charstring}) \)

**Return Value**

Result set

\( (type=\text{Bag of Vector}) \)
**queryBigtableLite**(query, params)

Generic parameterized GQL query interface that is optimized for small result sets. Attempts to query result set using one single connection. If this fails fallback to parallel processing using multiple connections.

**Parameters**

query: prepared SQL query statement  
(type=Charstring)

params: Vector of parameters for prepared statement  
(type=Vector)

**Return Value**

Result set  
(type=Bag of Vector)

---

**bigtableUpload**(file, relation, properties)

Bulk loading interface to upload bulk files of character separated values to Bigtable.

**Parameters**

file: bulk file  
(type=Charstring)

relation: the Bigtable relation  
(type=Charstring)

properties: Vector of properties to map the columns of the bulk file  
(type=Vector<Charstring>)
C. ODDSE reference

**bigtableRandomizedUpload(file, relation, properties)**

Optimized bulk loading interface to upload bulk files of character seperated values to Bigtable. Before bulk loading the bulk file is partitioned into chunks. These chunks are then randomly uploaded to the ODDSE server to improve load balancing.

**Parameters**

- **file**: bulk file  
  
  *(type=Charstring)*

- **relation**: the Bigtable relation  
  
  *(type=Charstring)*

- **properties**: Vector of properties to map the columns of the bulk file  
  
  *(type=Vector<Charstring>)*
**createBigtable**(*uri, name, properties, types, keyMembers*)

Creates a new Bigtable relation at the given location and instantly wraps it for usage in ODDSE.

**Parameters**

- **uri**: an ODDSE server location  
  *(type=Charstring)*
- **name**: the Bigtable relation name  
  *(type=Charstring)*
- **properties**: Vector of property names  
  *(type=Vector<Charstring>)*
- **types**: Vector of property types  
  *(type=Vector<Charstring>)*
- **keyMembers**: Vector of key members  
  *(type=Vector<Charstring>)*

**Return Value**

- true if succeeded  
  *(type=Boolean)*
### createBigtable\((uri, name, parent, properties, types)\)

Creates a new Bigtable relation at the given location using inheritance as described in the report and instantly wraps the relation for usage in ODDSE.

**Parameters**

- **uri**: the ODDSE server location of the parent  
  \((type=\text{Charstring})\)
- **name**: the Bigtable relation name  
  \((type=\text{Charstring})\)
- **parent**: the name of the wrapped parent Bigtable relation  
  \((type=\text{Charstring})\)
- **properties**: Vector of additional properties  
  \((type=\text{Vector<Charstring>})\)
- **types**: Vector of property types  
  \((type=\text{Vector<Charstring>})\)

**Return Value**

- true if succeeded  
  \((type=\text{Boolean})\)

### accessBigtable\(\text{uri}\)

Accesses and wraps all Bigtable relations from the AppEngine at the given location.

**Parameters**

- **uri**: an ODDSE server location  
  \((type=\text{Charstring})\)

**Return Value**

- true if succeeded  
  \((type=\text{Boolean})\)
createBigtable(name, uri)

Copies the schema of a wrapped Bigtable relation to a new App Engine and creates the relation there as well.

Parameters

- **name**: a wrapped Bigtable relation
  
  *(type=Charstring)*

- **uri**: an ODDSE server location
  
  *(type=Charstring)*

Return Value

true if succeeded

*(type=Boolean)*

analyzeBigtable(relation)

Analyzes the value range of each column of the given relation and stores the ranges in the ODDSE metadata.

Parameters

- **relation**: a wrapped Bigtable relation
  
  *(type=Charstring)*

buildHistogram(relation)

Builds a wavelet-based histogram on the key of the given relation to enable instant chunk generation from statistical data.

Parameters

- **relation**: a wrapped Bigtable relation
  
  *(type=Charstring)*
C. ODDSE reference

**saveOddseParams()**

Stores the current parameter set of the ODDSE wrapper to the metadata set maintained in Amos II.

**Return Value**

true if succeeded

(type=Boolean)

**reloadOddseParams()**

Reloads the parameter set of the ODDSE wrapper from the metadata set maintained in Amos II.

**Return Value**

true if succeeded

(type=Boolean)

**setOddseParam(name, value)**

Sets the parameter of the given name to a new value.

**Parameters**

name: parameter name

(type=Charstring)

value: new value

(type=Real, Boolean, Literal)

**Return Value**

true if succeeded

(type=Boolean)

**getOddseParams()**

Returns all ODDSE parameters

**Return Value**

ODDSE parameters

(type=Bag of <Charstring name, Real value>)
## C.1. ODDSE parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>optimizer_RecoveryInterval</td>
<td>Minimum interval between recovery actions (should be close to average latency)</td>
<td>Value: 5000 (type=Integer)</td>
</tr>
<tr>
<td>optimizer_ResAllocationInterval</td>
<td>Minimum interval between additional resource allocation</td>
<td>Value: 5000 (type=Integer)</td>
</tr>
<tr>
<td>optimizer_ResAllocationInterval_max</td>
<td>Upper limit for optimizer_ResAllocationInterval</td>
<td>Value: 50000 (type=Integer)</td>
</tr>
<tr>
<td>optimizer_QuotaFailure_blockTime</td>
<td>Default block time of resource usage in case of quota failures (will be adapted during processing)</td>
<td>Value: 1500 (type=Integer)</td>
</tr>
<tr>
<td>optimizer_QuotaFailure_blockTime_max</td>
<td>Upper limit for optimizer_QuotaFailure_blockTime</td>
<td>Value: 50000 (type=Integer)</td>
</tr>
<tr>
<td>chunkSize_Upload</td>
<td>Approximate number of rows per chunk for uploading (will be adapted during processing)</td>
<td>Value: 20 (type=Integer)</td>
</tr>
<tr>
<td>chunkSize_Upload_LowerLimit</td>
<td>Lower limit for chunkSize_Upload</td>
<td>Value: 20 (type=Integer)</td>
</tr>
<tr>
<td>chunkSize_Upload_UpperLimit</td>
<td>Upper limit for chunkSize_Upload</td>
<td>Value: 500 (type=Integer)</td>
</tr>
<tr>
<td>averageChunkSize_Upload</td>
<td>Approximate average realized chunk size on upload (a sliding window)</td>
<td>Value: 50 (type=Integer)</td>
</tr>
<tr>
<td>averageChunkSize_Upload_SLW</td>
<td>Size of sliding window to calculate average value for averageChunkSize_Upload</td>
<td>Value: 10000 (type=Integer)</td>
</tr>
</tbody>
</table>

*continued on next page*
## ODDSE parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Value</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>bigtableSubChunks</td>
<td>Number of sub-chunks to be build on the ODDSE server when calling the Datastore API (will be adapted during processing)</td>
<td>1</td>
<td>Integer</td>
</tr>
<tr>
<td>bigtableSubChunks_UpperLimit</td>
<td>Upper limit for <code>bigtableSubChunks</code></td>
<td>10</td>
<td>Integer</td>
</tr>
<tr>
<td>chunkSize_Query</td>
<td>Approximate number of rows for query chunks (will be adapted during processing)</td>
<td>500</td>
<td>Integer</td>
</tr>
<tr>
<td>chunkSize_Query_LowerLimit</td>
<td>Lower limit for <code>chunkSize_Query</code></td>
<td>100</td>
<td>Integer</td>
</tr>
<tr>
<td>chunkSize_Query_UpperLimit</td>
<td>Upper limit for <code>chunkSize_Query</code></td>
<td>1000</td>
<td>Integer</td>
</tr>
<tr>
<td>chunkSize_Query_minChunkFactor</td>
<td>This factor defines the interval size of accepted target chunk sizes <code>[minFactor*size, size]</code></td>
<td>0.8</td>
<td>Real</td>
</tr>
<tr>
<td>averageChunkSize_Query</td>
<td>Approximate average realized chunk size on query processing (a sliding window)</td>
<td>500</td>
<td>Integer</td>
</tr>
<tr>
<td>averageChunkSize_Query_SLW</td>
<td>Size of sliding window to calculate average value for <code>averageChunkSize_Query</code></td>
<td>10000</td>
<td>Integer</td>
</tr>
<tr>
<td>threads</td>
<td>Default number of used threads, respectively server connections (will be adapted during processing)</td>
<td>4</td>
<td>Real</td>
</tr>
<tr>
<td>threads_LowerLimit</td>
<td>Lower limit for threads</td>
<td>4</td>
<td>Real</td>
</tr>
<tr>
<td>threads_UpperLimit</td>
<td>Upper limit for threads</td>
<td>50</td>
<td>Real</td>
</tr>
</tbody>
</table>

*continued on next page*
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Value</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>thread_minPoolFactor</td>
<td>This factor defines the interval [core thread pool size, max thread pool size (threads)]</td>
<td>0.8 (type=Real)</td>
<td></td>
</tr>
<tr>
<td>thread_defaultPriority</td>
<td>Default priority of Java threads</td>
<td>5 (type=Integer)</td>
<td></td>
</tr>
<tr>
<td>thread_highPriority</td>
<td>High priority for sequential resume or cursor queries</td>
<td>8 (type=Integer)</td>
<td></td>
</tr>
<tr>
<td>terminationWaitTime</td>
<td>Maximum wait time for thread termination in minutes</td>
<td>8 (type=Integer)</td>
<td></td>
</tr>
<tr>
<td>filesystemSeparatorChar</td>
<td>Default separator used for character seperated files (bulk loading)</td>
<td>'</td>
<td>' (type=Literal)</td>
</tr>
<tr>
<td>autoStoreParams</td>
<td>Automatically store the adapted parameter set of the ODDSE wrapper in Amos II II after processing</td>
<td>True (type=Boolean)</td>
<td></td>
</tr>
</tbody>
</table>