Ensuring BASE Consistency in a CRUD Middleware Layer for Heterogeneous Databases

Master Thesis

Christian Klim Hansen and Nedim Cokljat
Department of Computer Science
Selma Lagerløfs Vej 300
Aalborg, Denmark
{ckha10, ncoklj10}@student.aau.dk

ABSTRACT

Research into heterogeneous database systems and alternative database models has become more and more relevant as the amount of stored data continues to increase. Specialized database models are created to perform efficiently when performing a certain workload. Graph databases are well suited for traversal queries, such as finding the shortest path. The typical information queried for in graph databases is meta data, such as finding how nodes are related to each other trying to find a pattern in the graph structure. Relational databases are more powerful when it comes to querying for data about specific entities, since over the years a lot research has been put into optimizing indexing in relational databases. But when it comes to finding associations between entities joining of tables is required, and joining complex associations can be slow, and that is where graph databases outperform relational databases since they do not have to join.

Bridge-DB is a fault tolerant middleware system that connects multiple different database models and enables users to read and write to externally connected heterogeneous databases while ensuring eventual consistency and availability. Bridge-DB is tested with an external Neo4J and PostgreSQL database using full redundancy, but is designed to be a general framework for working with multiple heterogeneous databases. The tests illustrate the need for Bridge-DB by showing how different query types are executed faster and slower depending on the selected database model. Our experiments further document, that Bridge-DB upholds the BASE properties while supporting all create, read, update and delete operations.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems; H.2.5 [**Database Management**]: Heterogeneous Databases

General Terms

Data Management, Distribution, Experimentation, Design

Keywords

Middleware, Database Consistency, Heterogeneous Database, Concurrency Control, Distributed Query Management, BASE Properties, CAP Theorem

1. INTRODUCTION

The recent high demand for support of big data has led to an increase in research into alternative database models. Databases are essential to almost all applications we use today, and researchers are trying to find efficient ways to both access and process the stored data [17, 13, 1]. In terms of research into database models we see a divide between the relational model and the NoSQL movement. Many database models exist and they are all different in minor or major ways. These differences are seen in the structure of the data, how the data is queried, and how it is stored on disk etc. Some databases such as graph databases are better at running traversal queries whereas other databases, like relational databases, are better at read queries. However, we do not believe that the solution is a single database model that incorporates all features from existing databases. This solution could lead to a very complex database model with a large amount of features, that would not necessarily have high performance. Instead we are convinced that a middleware layer system, which connects different database models and adds a level of abstraction for the users to interact with multiple heterogeneous databases, is the proper solution to the problem at hand.

In Bridge-DB v.1.0[1] [10] it was shown that the middleware solution only added a small amount of overhead to queries compared to running the test queries directly on PostgreSQL or Neo4J. It also contains a heuristic optimizer that automatically chooses where to send read queries. Full replication between the databases was ensured, however, insert, update, and delete queries were not supported in Bridge-DB v.1.0.

This project focuses on developing Bridge-DB v.2.0 to support insert, update, and delete queries alongside weak consistency between the database models connected to Bridge-DB. When implementing the CRUD operations in a distributed system using full replication of data on all connected external databases a major topic to consider is data consistency. Therefore, how consistency is done in Bridge-DB is extensively elaborated on in this project. Managing fault tolerance is another topic which is relevant to investigate. Since if an external database is disconnected from Bridge-

[1]Throughout the article the first version of Bridge-DB is referred to as "Bridge-DB v.1.0" and the new version is referred to as "Bridge-DB v.2.0".

DB, consistency still has to be ensured when the connection to the external database is reestablished. Another problem that has to be considered when a database is disconnected is the availability of other data sources.

Bridge-DB should support the BASE properties, because that enables users to query the system even when it is not fully consistent and still get reliable results. An analysis of the ACID and BASE properties is presented in Section 2 as the background for our implementation, as well as a discussion of the CAP theorem, as presented by Eric Brewer [5]. In Section 3 related work done on heterogeneous databases, distributed databases, and optimized systems are elaborated on. In that section we also talk about what makes our system unique compared to other similar systems. The Bridge-DB v.2.0 architecture and important design considerations are presented in Section 4. The changes that are made to the architecture to get from Bridge-DB v.1.0 to Bridge-DB v.2.0 are also illustrated in this section. Important parts of Bridge-DB which have been implemented are shown in Section 5. Section 6 presents our test results, how the experiments were conducted, and a discussion of the findings is given. Lastly the project is concluded in Section 7 and possible future work is presented.

2. BACKGROUND

2.1 ACID, BASE and the CAP Theorem

According to the CAP (Consistency, Availability, Partition Tolerance) theorem first introduced in 2000 by Eric Brewer [5] and later proven to be true by Seth Gilbert and Nancy Lynch in 2002 [7], it is only possible to have two of the three properties in a system. The CAP theorem specifically addresses network partition tolerance, which is necessary in the case where large amounts of data are stored on different server sites. If one server is malfunctioning, the users should still have access to the data on other servers. The choice then becomes a choice between consistency or availability. That said it is a bit misleading, because a system can have all three properties, if it is designed to change between focusing on consistency and availability in certain situations [4]. Before we continue, note, that we define a database transaction to be either a read or write transaction consisting of one or multiple queries. These queries may perform any CRUD operation. The only restriction enforced is that a read query cannot be in the same transaction as a write query.

The three properties in CAP are as follow [4]:

Consistency means that one copy of the data is up to date.

Availability means that when issuing a request it will eventually return a result.

Partition Tolerance means that if the network is partitioned such that communication between the involved servers cannot happen then messages get lost, but the system should keep running.

Consistency in CAP is often misunderstood because people tend to associate it with the "C" in the ACID (Atomicity, Consistency, Isolation, and Durability) properties, which is a model for ensuring consistency between databases [9, 8]. `Atomicity` ensures that all queries in a transaction either succeed or fail. If one query fails the whole transaction fails which causes a rollback. Otherwise, if all queries succeed the transaction commits. `Consistency` ensures that no transaction will bring a database into an invalid state based on predefined rules of the database. These rules include, but are not limited to, constraints, data types, cascading deletes, and foreign keys. `Isolation` ensures that all transactions happen independently, meaning that one transactions does not interfere with another. This is relevant when considering concurrent evaluation of queries. If two transactions evaluate concurrently the resulting database state must be the same as if the two transactions were evaluated sequentially. `Durability` guarantees that results completed and committed under the restriction of the previous three properties are not lost in the case of failure. This involves storing data in non-volatile memory.

Based on the CAP Theorem and the fact that the need to store large amounts of data will only increase, alternative consistency models to the ACID properties were introduced. These models are also called weak consistency models, and one of the most well known weak consistency model is the BASE (Basically Available, Soft state, and Eventual consistency) properties [16].

The BASE properties were defined by Eric Brewer and his team in the late 1990s. One among other reasons for creating BASE was to give designers of computer systems a new way to think about achieving high availability [4]. However, instead of clearly defining the properties Brewer and his team provided an overview of the differences between the ACID and BASE properties [5].

The BASE properties focus on ensuring availability (`Basically Available`) and that all changes to one database will eventually propagate to all replications of that data (`Eventual consistency`). `Soft State` is not a clearly defined property, and it is debated what it actually means. It often refers to the fact that the database will do something without receiving input from the user. One interpretation is that since all databases in the system will eventually become consistent, the state of the databases will change without user input. However, this interpretation makes the soft state property a redundant property of the BASE properties, since eventual consistency is already a property of BASE. Another interpretation refers to the fact that data can expire if the user does not maintain the data. This would be a valid interpretation if all databases acted this way, however, not all do. Eric Brewer admits that the acronym BASE is a bit contrived, and that the same thing was true with the ACID properties, which he discussed with Jim Gray [6].

2.1.1 *Implementing BASE*

There exist various different ways of implementing BASE consistency into a system, but the main differences consist of where the initial update is performed (predefined master node or arbitrary location), where users get data (from master node or arbitrary location), and whether updates are performed synchronously or asynchronously. Improved consistency can be gained by performing updates on a predefined master node and letting users always read from the

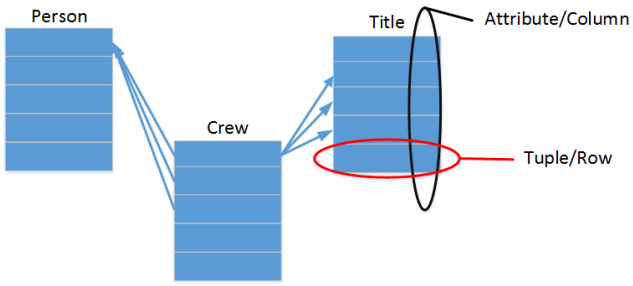

Figure 1: Relational Model: 3 tables where a person can be a crew member on multiple different movies

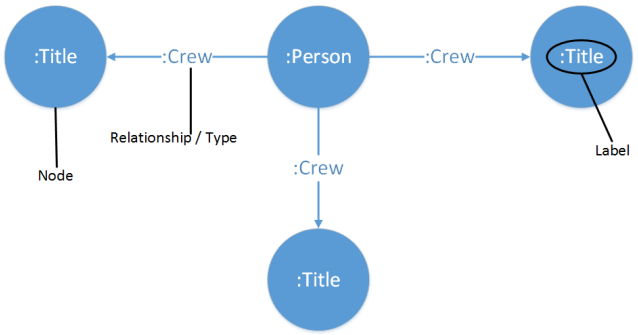

Figure 2: Neo4J version of Figure 1

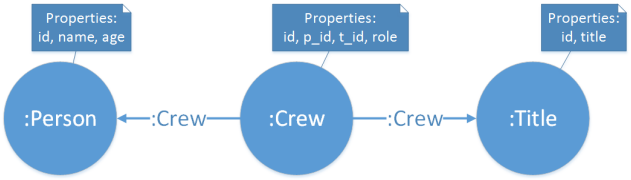

Figure 3: Structure of Nodes in Neo4J

master node, however, this will significantly increase latency. Always updating on a predefined master node and propagate the update to all replications and allowing users to access arbitrary databases could result in users reading inconsistent data, but improve on latency issues [4].

2.1.2 *Extending the CAP Theorem*

Although the CAP Theorem is a common way of looking at consistency, Daniel J. Abadi from Yale University introduced a possible extension to the theorem in 2012 called PACELC (pronounced 'pass-elk') [2]. In the article, Daniel J. Abadi introduces the tradeoff between consistency and latency and argues that choosing a weaker consistency model is often not only a result of wanting increased availability, but also an interest in ensuring low latency. Latency is an issue in terms of user experience. Studies show that even an increased latency of as small as 100 ms has an effect on whether customers will return to a web site in the future [2].

2.2 PostgreSQL and Neo4J

The relational model and the graph model capture the same information about data, but in very different ways.

2.2.1 *Relational-Graph Conversion*

In order to support both models, we need a way to convert the relational model into a graph model. We have not developed an automated tool for this task, but such a tools would be interesting for future research. Currently, the conversion from relational to graph model is based on considerations in terms of foreign keys as well as id mapping. The problem of id mapping is described in detail in Section 4.3. We use an example based on the IMDb data set described in Section 6.1.

As illustrated in Figure 1, data in PostgreSQL is stored in tables with attributes. Entries into a relational table are referred to as either rows or tuples. In Neo4J attributes are called properties and are stored in nodes or edges between nodes. These nodes and edges correspond to tuples in PostgreSQL. PostgreSQL identifies tables using table names. Neo4J identifies tables by either the node label or the edge type. Edges between nodes in Neo4J are called relationships and is Neo4J's way of representing foreign key relations between tuples in a relational database without creating a separate node. This can be seen in Figure 2 where one person has been a crew member on three different movie titles. Here the `Crew` table is illustrated as an edge instead of a node. This works fine in this example, since the `Crew` table only has two foreign keys. However, since it is not possible to create an edge between more than two nodes in Neo4J, and tables in relational databases may contain more than two foreign keys. So in order to support more than two foreign keys we changed how we present tables with foreign keys from as illustrated in Figure 2 to Figure 3. All tuples in PostgreSQL are represented as nodes in Neo4J with the same properties/attributes as the tables in the PostgreSQL database.

Neo4J is a directed graph, so we define a standard way of representing foreign key references in a directed graph. We use Figure 3 as an example. In this project we impose the logic, that an edge between two nodes always points from the node that references the other node, and that the label of that edge is the same as the label of the node from which the edge points from. On Figure 3 the `Crew` node references both the `Person` and `Title` nodes, as indicated by the foreign keys `p_id` and `t_id` on the `Crew` node. This means that the edges must start from the `Crew` node and point to both the `Person` and `Title` node, and that the edge type must be `Crew`.

2.2.2 *Internal Database Schema*

In order for Bridge-DB to get information about the schema of the connected database, we need an internal representation of the database schema. This must be stored locally on Bridge-DB, and changes to the external database schema must be reflected in the internal representation to maintain consistency.

The internal database schema (IDS) is our way of representing the database schema of the externally connected databases. It is used by the translators for translating the Bridge-DB Query Language into either Cypher or SQL and assumes unique table names. It is designed as a JSON file

and based on the relational model. It contains information about constraints and a list of all attributes. Listing 1 illustrate the basic structure of the IDS using the `Crew` table as an example. Information about datatypes are not included in this version of the IDS, since we are only working with two datatypes, numbers and strings, and Neo4J does not make a distinction between other datatypes otherwise supported by PostgreSQL e.g. date and JSON.

Listing 1: Basic Structure of Internal DB schema

```
{
   "Crew" :
   {
      "primary_keys" : [ 'id' ],
      "foreign_keys" :
      {
         "person_id" : "Person.id",
         "movie_id"  : "Title.id"
      },
      "not_null" : [ 'id' ],
      "unique"   : [ 'id' ],
      "default_value" :
      {
         "note" : "default_value"
      },
      "standard_attr" : [ 'id', 'person_id',
          ... ]
      "counter" : max('id')
   },
   "Title" : { ... }
}
```

The `primary_key` can be any numeric datatype as long as it is unique. The `foreign_key` is a key-value pair, where the key is the foreign key attribute and the value describes the table and attribute that the foreign key references. Additionally we have added the constraints `not_null`, `unique`, and `default_value`. The primary key includes in both `not null` and `unique`. The default value is also a key-value pair consisting of the name of the attribute as the key and the default value as the value. Lastly, all attributes are also stored in `standard_attr` and the maximum primary key value is stored in the `counter` which is for insert queries. The primary focus of Bridge-DB v.2.0 is to support all CRUD operations as well as the BASE properties. This means that composite keys and string primary keys are not supported in version 2.0.

3. RELATED WORK

MOCHA [13] is a middleware system which allows user defined types and operations. It is a self-extensible system, which means, that when a user adds new functionality to the system, e.g. some new data type or operation,then it is automatically distributed to all connected server sites if the new functionality is required for the execution of a query. In MOCHA they increase the performance of the system by shipping either the query, the data, or a hybrid version of these methods. When shipping the data, the data is moved from the site containing it to the site requesting the data. When a query is shipped, the operators of that query are executed on the data sites where the data resides. MOCHAs first optimization factor is the network, since it is a shared resource and moving the data in large-scale environments typically impose a major performance bottleneck. The approach they use for improving network performance is pushing *data-reducing* operators to be evaluated on the remote data sites and *data-inflating* operators to be evaluated on the client site [13]. The *data-reducing* operators are operators that reduce the size or amount of tuples returned in the result. Whereas the *data-inflating* operators are the operators which inflate the data value or present them in many forms, projections, levels of detail etc. [13]. MOCHA tries to improve performance by shipping the query or the data, and it chooses the best data source for the query. But one of the problems they do not solve is ensuring the BASE properties, as done by Bridge-DB. They also do not look at what kind of problem has to be solved, e.g. if it is a reachability problem, MOCHA does not take that into account when choosing the data source. The MOCHA prototype supports object-relational databases such as Oracle and Informix [12] as well as XML repositories and file systems [13]. Bridge-DB implements a heuristic optimizer that automatically chooses the most compatible database based on the query type and available databases.

Another type of systems related to Bridge-DB are federated database systems which consist of multiple independent autonomous component databases which appear to function as one entity. These component databases are self-sustained, which makes them independent of other systems to function properly [11]. A federated system may support both heterogeneous and homogeneous databases. For the latter case the federated system is used to distribute the load of very large databases. A federated system works by enabling the components interact and share information in a reasonable degree. This is done by each component defining what information to share with other components and with which other components [11]. The difference between federated systems and Bridge-DB is that in Bridge-DB the user has full control of the system, whereas in a federated system the user does not have full control. Therefore a federated system does not solve the problem that we face, since we want the user to have full access to the external databases. Hence a closer relative to Bridge-DB is Distributed Database Management Systems (DDBMS).

These systems have a central unit which controls all the connected databases to the network, and each database is accessed through the central unit. In DDBMSs the user has full control but the problem with these systems is, if they control heterogeneous databases then all the connected databases usually use different query languages. The user is then required to learn several languages to be able to use the system, if an additional overlay is not provided. Therefore DDBMSs do not solve the problem of easing the use of heterogeneous database system which is one of the goals of Bridge-DB.

GRAPHITE [14] is a newly proposed graph traversal framework that adds traversal operators to an RDBMS. The system chooses the best traversal operator based on a graph topology and traversal query. To handle graph queries efficiently a property graph model and a common storage infrastructure that holds the property graph is included in the system. The authors show that a single database engine can efficiently handle both relational and graph operations. But still this does not solve the problem which we face. With their framework they solve the problem of relational data-

bases not being able to efficiently run traversal queries. The problem that we want to solve is to ease the use of a multi heterogeneous database system. This is partly solved by providing a single language for all connected external databases. Additionally Bridge-DB should support all CRUD operations as well as the BASE properties to ensure eventual consistency and high availability. None of the above mentioned systems solve the problems solved by Bridge-DB.

4. DESIGN

4.1 Architecture

The Bridge-DB v.2.0 architecture has changed compared to Bridge-DB v.1.0 [10]. We will first quickly explain the architecture of Bridge-DB v.1.0 to provide an easy overview of the changes. Then the new architecture will be presented as well as any additional design considerations such as id mapping, and the improvements made to the Bridge-DB Query Language (BQL).

4.1.1 Bridge-DB v.1.0

Bridge-DB is based on a three layer architecture consisting of the interaction interface, Bridge-DB, and external databases as illustrated in Figure 4. The architecture is designed to enable easy integration of new database systems using a modular approach, where users can add and remove their own modules. Bridge-DB works as an extra abstraction layer between the user and the external databases. Queries are sent to the external heterogeneous databases for execution based on a heuristic optimizer and only read queries are supported. The heuristic optimizer was created based on the experiments performed in our previous work [10]. The results showed that graph traversal and reachability queries were faster to execute on Neo4J, while regular 'SELECT-FROM-WHERE' read queries performs best on PostgreSQL.

The user writes a query in the Bridge-DB Query Language (BQL), which is then sent to Bridge-DB in JSON format. Bridge-DB transforms the JSON object into our own Query object and based on the optimizer sends the query to the most compatible connector. The connector then translates the Query object into either SQL or Cypher, connects to the external database, sends the query string, and receives a result. This result is finally returned back to the user. MongoDB was initially part of the Bridge-DB architecture. It was meant to handle id mapping but was never implemented. The configuration file stores information about the registered databases, but was also never used during testing.

4.2 Bridge-DB v.2.0

Bridge-DB v.2.0 adds a lot of improvements to Bridge-DB v.1.0, while maintaining the original module based approach, meaning Bridge-DB is still designed for easy integration of new databases using user customized modules. Figure 5 illustrates the new architecture. Bridge-DB v.2.0 focuses on efficient handling of database transactions sent by multiple users while maintaining availability and eventual consistency between the external databases by supporting the BASE properties.

Queries are written by users and all queries sent simultaneously by one user are grouped as one transaction. Currently this is done in PHP using BQL. To access BQL the

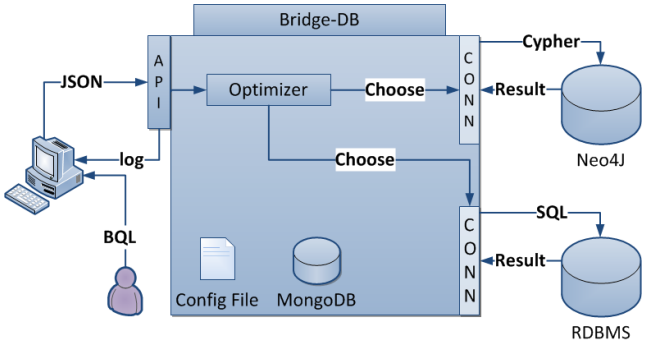

Figure 4: Bridge-DB v.1.0 Three Layer Architecture

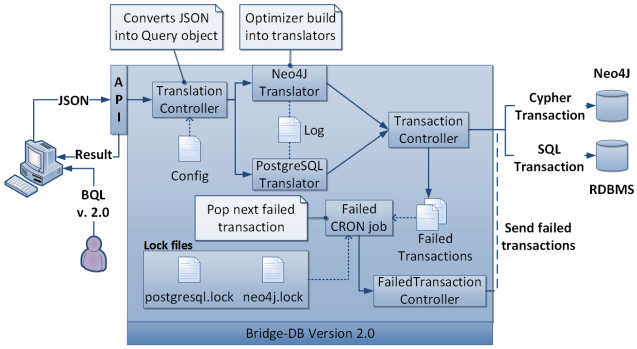

Figure 5: Bridge-DB v.2.0 Architecture

user must include the QueryBuilder class and the REST Client. It should be noted, that read and write queries are not allowed in the same database transaction in BQL, but multiple write queries in the same transaction is allowed. Write queries can be used to either `insert`, `update`, or `delete` data in the external databases or to `alter` the underlying database schema. The BASE properties ensures, that all writes queries performed on one database will eventually be executed on the remaining databases, thereby guaranteeing eventual consistency.

This project defines a database transaction in BQL as a single instance of the QueryBuilder class. A database transaction consists of one or more queries, which the user inserts by calling BQL functions. The user is free to insert queries into a transaction until it is closed. A transaction is defined to be closed when the user calls the `send()` BQL function. A closed transaction can not receive anymore queries from the user. Note, that the queries will be executed in the added order. The user is therefore responsible for adding the queries in the desired execution order. Listing 2 shows an example of how a user could create a transaction. A new instance of the QueryBuilder class is created on Line 1, adds an `insert` query on Line 2 followed by an `update` query on Line 3. The transaction is closed on Line 4. Assuming that the transaction does not fail, the `insert` query will be executed before the `update`, because of the order the queries were added to the transaction. If the transaction fails, Bridge-DB is designed to be fault tolerant. Fault tolerance is discussed in Section 4.2.1.

Listing 2: Example of user creating a new database transaction

```
$dbTransaction = new QueryBuilder($bridgeDbIP);
$dbTransaction->insert("Person", "id, name, age, gender", "2, Jens, 25, male");
$dbTransaction->update("Person", "age", "26", "Person.name = 'Jens' AND Person.age = 25");
$dbTransaction->send();
```

After closing the transaction, the query is sent to the Bridge-DB `TranslationController` as a JSON object, where it is converted into our own Query class object. Bridge-DB v.1.0 only supports read queries, which means that the Query object is passed to both the Neo4J and PostgreSQL connector, where the query is evaluated by the heuristic optimizer for compatibility with the connected database. In Bridge-DB v.2.0 the Query object contains a queue of queries in the order the user adds them. The connectors are replaced with translators that receive the query queue and translates the queue into either SQL for PostgreSQL or Cypher for Neo4J. The heuristic optimizer of Bridge-DB v.1.0 is build into the translator, but nothing new has been added other than supporting the basically available property. For example, if a connection to PostgreSQL can not be established, the optimizer makes sure, that read queries are executed on Neo4J instead, even though Neo4J would execute the query slower than PostgreSQL. Otherwise, the query will only be translated to the most compatible database. Most compatible in this project means the database, that would execute the transaction the fastest based on the results found in our previous work [10]. The translators return an array of SQL and Cypher queries.

Both arrays are first logged in a log file, that stores information about committed and failed transactions as seen in Listing 3. In this example we try to insert five new persons into the database. The Neo4J transaction committed, however, the PostgreSQL failed. After the transactions has been logged, the arrays are passed directly to the `TransactionController`.

This is different from Bridge-DB v.1.0. In the previous version the connectors were responsible for both translating the Query object into Cypher and SQL, and sending the query to the external database. However, since we are working with a queue of queries that needs to be evaluated simultaneously, we added the `TransactionController`. This controller receives the Neo4J and PostgreSQL query arrays simultaneously as a JSON object. The SQL and Cypher queries are evaluated on their respective external databases and if all queries succeeded the transaction commits on the external databases. Otherwise the transaction failed and rolls back all changes resulting from the queries. The implementation is described in Section 5.2. The result is returned back through the previous steps until it reaches the user. Note, that the TransactionController stores failed transactions in database specific files, meaning that failed PostgreSQL transactions are stored in the file `failed_postgresql.json` and failed Neo4J transactions are stored in `failed_neo4j.json`. If a write transaction fails, the database is locked by creating a lock file. A

Listing 3: Example of log file storing committed failed and pending transactions

```
{
  "pending": [],
  "committed": [
    [
      " MERGE (n:Person { id : 1, name : 'Sara', age : 23, avg : 8.9, sex : 'female'});",
      " MERGE (n:Person { id : 2, name : 'Jens', age : 25, avg : 10.3, sex : 'male'});",
      " MERGE (n:Person { id : 3, name : 'Andreas', age : 24, avg : 6.7, sex : 'male'});",
      " MERGE (n:Person { id : 4, name : 'Flemming', age : 24, avg : 7, sex : 'male'});",
      " MERGE (n:Person { id : 5, name : 'Louise', age : 24, avg : 8.3, sex : 'female'});"
    ]
  ],
  "failed": [
    [
      "INSERT INTO Person(id, name, age, avg, sex) VALUES(1, 'Sara', 23, 8.9, 'female');",
      "INSERT INTO Person(id, name, age, avg, sex) VALUES(2, 'Jens', 25, 10.3, 'male');",
      "INSERT INTO Person(id, name, age, avg, sex) VALUES(3, 'Andreas', 24, 6.7, 'male');",
      "INSERT INTO Person(id, name, age, avg, sex) VALUES(4, 'Flemming', 24, 7, 'male');",
      "INSERT INTO Person(id, name, age, avg, sex) VALUES(5, 'Louise', 24, 8.3, 'female');"
    ]
  ]
}
```

CRON job is responsible for constantly getting the next failed transaction and trying to execute the query on the correct database using the `FailedTransactionController`. This controller is very similar to the regular TransactionController, but ignores lock files. If the transaction is committed it is removed from the failed transaction file, however, if it did not commit, the CRON job will try again. The lock file will only be deleted when no more transaction exist in the failed transaction file. The CRON job and `FailedTransactionController` ensures eventual consistency by executing the failed transactions in the correct order and makes sure to unlock any locked external database, that no longer has any failed transactions by deleting the lock file.

4.2.1 Fault Tolerance

Failures is unavoidable when working with any kind of computer system that has some level of complexity and databases systems are no exception. It is therefore important that Bridge-DB is able to handle transaction failures correctly depending on the failure. In this project we consider

the following failures:

- One or more queries of a database transaction fail
- Connection to one database is lost before transaction can be committed
- Connection to one database can not be established

Transaction Failure. This can for example occur if the user tries to insert a new tuple and provides a primary key that already exists. A failure of one query in a transaction is handled as a failure of the entire transaction. Nothing is committed to the external databases and the error is reported back to the user.

Connection Loss. This is a mid process error which occurs because the connection to an external database is lost while evaluating queries in a transaction. This means, that the transaction is unable to both execute the next query and commit even if none of the queries in the transaction would result in a failure. Bridge-DB then locks the external database where the connection was lost. This restricts future write transactions by creating a database specific lock file, i.e. `postgresql.lock` or `neo4j.lock`. It is still possible to perform read transactions on locked databases.

Bridge-DB stores the failed transaction in the separate failed transaction files (FTF) `failed_postgresql.json` and `failed_neo4j.json`. A CRON job would then periodically check the FTF and try to execute all failed transactions in the FTF in the correct order. This approach allows us to report the failure back to the user. Although the connection may be lost to one of the external databases, the transaction can still be executed on all other connected databases. When the connection is reestablished the transactions from the FTF can be executed on the reconnected database. This solution ensures eventual consistency and the updated heuristics optimizer ensures the remaining connected databases are still available and that read queries are executed normally.

Missing Connection. This failure is similar to the previous, but occurs in the TransactionController before the queries of the transaction are executed. If a connection to one or more databases can not be established, the transaction will fail. Note, that a connection may still be established to another database, so a write transaction may still be committed on Neo4J even though the connection to PostgreSQL could not be established.

This failure is handled as if the connection was lost between query executing. The external database is locked using a database specific lock file, and the failed transaction is stored in a database specific FTF. The error is reported back to the user. The CRON job recovers the failed transaction when connection is reestablished.

4.3 ID Mapping

In Bridge-DB v.1.0, id mapping between Neo4J and PostgreSQL was intended to be handled by MongoDB, but was never implemented.

Since all nodes in Neo4J are automatically assigned a unique id (accessed by using `id(n)` where n is a node), and primary keys in PostgreSQL are only unique for each table, id mapping between the two databases is needed in order to enable querying for specific `id`s. This is done by adding the primary key and foreign keys of PostgreSQL tables as properties on nodes as seen on Figure 3 on page 3. Here the `Crew` node has the properties `id`, which is the primary key of the PostgreSQL `Crew` table, `p_id`, which is a foreign key to the `Person` table, `t_id` which is a foreign key to the `Title` table, and a `role` property describing the role the person had on a given title.

Because id mapping is implemented by adding additional properties to the Neo4J nodes, insertion of new nodes/tuples into both Neo4J and PostgreSQL needs to be considered. Inserting a new tuple into PostgreSQL does not always require the user to input the primary key if PostgreSLQ auto-increments the primary key. Note, that not all tables have auto incremented primary keys, however, we leave it to the user to provide a unique primary key if auto-incremented primary keys are not implemented. This does not work for Neo4J, since the PostgreSQL primary key property of Neo4J is depended on PostgreSQL. A solution to this problem would be to handle all insertion queries on PostgreSQL first, get the new primary key from PostgreSQL and use that primary key as input to Neo4J. This would possible lead to a slower execution time and concurrent processing is also not available if using this approach.

Another more elegant solution is to store the highest primary key on each table in the internal database schema (IDS) and use that as input for both Neo4J and PostgreSQL, if the user does not manually provide a primary key. This approach was implemented as it allows us to perform queries on the external databases concurrently. However, there are more considerations when adding more features to BQL. The following section will go into detail about the added features and the consequences of implementing them.

4.4 Improving BQL

BQL is developed to be similar to SQL while supporting special functions for graph databases, such as graph traversal and reachability. Improvements to BQL extend beyond adding additional functionality, since introducing write queries creates a new set of challenges. To make this paper more standalone a list of the previous BQL functions is available in Appendix A, but can also be found in our previous work [10]. Table 1 lists the added functions in BQL v.2.0.

4.4.1 Insert, Update, and Delete

`insert()` takes as parameter the table we want to insert a new tuple into, an array of columns, and an array of values. The QueryBuilder will display an error message to the user if they try to insert into more than one table or the number of columns and values does not match. The user can insert a tuple into a table that either does or does not have a

Name	Input	Return
rename_table	@param: table:string, newTable:string	@return self
rename_attr	@param: table:string, oldAttr:array, newAttr:array	@return self
remove_attr	@param: table:string, attrs:array	@return self
insert_attr	@param: table:string, attr:string, datatype:string, constraints:array, defaultValue:mixed	@return self
update	@param: table:string, columns:array, values:array, condition:string	@return self
insert	@param: table:string, columns:array, values:array	@return self
delete	@param: table:string, condition:string	@return self
send		@return json:object

Table 1: Added functions to BQL in Bridge-DB v.2.0

Listing 4: Two different ways of inserting

```
1 $dbTransaction = new QueryBuilder($bridgeDbIP);
2 $dbTransaction->insert('Person', ['id', 'name',
      'age'], [1, 'Jens', 25]);
3 $dbTransaction->insert('Person', ['name', 'age'
    ], ['Jens', 25]);
```

foreign key constraint. This context does not matter in the translation to SQL, but if a table contains a foreign key in Neo4J, an edge between two nodes needs to be created. The internal database schema (IDS) described in Section 2.2.2 helps the translators with the translation by detecting foreign keys in queries.

An additional challenge is that we need to include the PostgreSQL primary key into Neo4J as well to handle id mapping as discussed in the previous section. Listing 4 illustrates two different ways the user can insert a new person named 'Jens' into the database. The difference consists of whether the user provides the primary key, which in this case is `id`, or does not. The current maximum counter for the primary key is stored in the IDS. We impose the logic, that if the user provides a manual primary key as done on Line 2, it must not be smaller than the current counter value, since auto-increment solutions only increment the primary key and never fill in eventual primary key holes created by deleting tuples. If the user manually inserted a valid primary key, the translation proceeds and the new counter value is updated in the IDS. However, this would allow the users to manually insert primary key values, that are much larger than the current counter value, thereby skipping a lot of values. We therefore restrict how much larger the manual inserted primary key can be compared to the current counter value. The restriction is currently set to 10, but can easily be changed if necessary. Note, that if the user inserts as done on Line 3, the counter plus one will be used as the primary key, and updated at the same time.

Listing 5: Example of delete in BQL

```
1 $dbTransaction->delete('Person', 'name = Jens
      AND age = 25');
```

The `delete()` function takes a table and a condition as parameters. The transaction in Listing 5 deletes all tuples from the `Person` table that fulfill the condition. Processing the condition is a complicated task, which requires two additional function calls in the QueryBuilder in order to format the condition in the right way. One challenge that the delete function imposes is cascading deletes. BQL supports both cascading and non-cascading deletes, but must be setup before hand. Note, that in Neo4J it is not possible to delete a node that still has edges to other nodes, so all edges from a deleted node must also be deleted.

Listing 6: Example of update in BQL

```
1 $dbTransaction->update('Crew', 'p_id', '4', '
      Crew.id = 1');
```

The `update()` function is one of, if not, the most complicated new function introduced in BQL v.2.0. It might seem similar to the rename, remove, and insert attribute functions, but we distinguish between update function that change data, like `update()`, and functions that change the database structure, like `rename_table()`. `update()` takes a table, an array of columns, and an array of values, and lastly a condition as parameters. Only one table can be updated at a time, and the number of columns and values must match. The processing of the condition is the same as in `delete()`. One of the challenges imposed by the update function is handling updating of foreign keys.

In Listing 6 we are updating the foreign key `p_id` for the crew tuple/node with `id=1`. In PostgreSQL this is a simple update, but in Neo4J we need to delete and create new edges between nodes. In Neo4J, if a `Crew` node A has one foreign key to a `Person` node B and the user updates the foreign key to point to the new `Person` node C instead of B, then the edge from A to B must first be deleted, C must be found, an edge from A to C must be created, and the foreign key property of A must be updated to be the same as the primary key property of C. This involves a lot of look up in the IDS for the Neo4J translator and can possibly take a large amount of time depending on the size of the data set, because C needs to be found before the edge can be created. We must also ensure that if the user updates an attribute having the

unique constraint, that the new value is indeed unique. It is expensive in Neo4J to check for uniqueness, since it would involve traversing the entire graph. This means, that in these unique update cases, it would be more efficient to send the query to PostgreSQL for validation before Neo4J, since PostgreSQL would immediately fail if an existing value was provided.

4.4.2 *Updating the Database Schema*

Besides the traditional insert, update, delete functions, BQL v.2.0 provides functions that can change the database schema. The users can rename both table and attribute names as well as add and remove attributes. The changes to the database schema must be committed on all external databases before the IDS is updated. Only the function `insert_attr()` is none trivial to implement.

Listing 7: Example of inserting either new not null or unique attribute in BQL

```
1 $dbTransaction->insert_attr("Person", "height",
      "int", ["NOT NULL"], 150);
2 $dbTransaction->insert_attr("Person", "cpr", "
    int", ["UNIQUE"]);
```

The purpose of this function is to add a new column to an existing table, that already contains data. The function takes the table name, attribute name, datatype, an array of constraints, and an optional default value as parameters. The QueryBuilder must do a lot of checks before the query is validated. Only one table, attribute, and datatype can be included, and the constraints must be expressed as an array, e.g. `["NOT NULL", "FOREIGN_KEY:Person.id"]`. In this example, the foreign key references the id attribute on the Person table. Additional checks must be made. If the attribute is not allowed to be null, a default value is required, otherwise the new column would automatically be populated with null values in PostgreSQL. If the attribute must be unique, a default value is not allowed, since a default value contradicts the notion of uniqueness. This leads us to the conclusion that we can not insert a new attribute into a table with both the not null and unique constraints assuming the table already contains data. Two different examples of inserting a new attribute into a table is shown in Listing 7. Line 1 insert the attribute `height` into the `Person` table as an int with the `NOT NULL` constraint and a default value of 150. Line 2 does the same with the attribute `cpr` using the `UNIQUE` constraint and omitting a default value. Note, that the primary key is always both `NOT NULL` and `UNIQUE`, but this was defined before data was added to the table. The QueryBuilder does not check for valid datatypes and does not compare the default value with the provided datatype. The default value will be interpreted as provided by the user, e.g. the value ”0” is a string and the value 0 is an integer.

These added functions are currently available for everyone using the system. Ideally the users would log into Bridge-DB through a user interface and have different access permissions. Only the database administrator should be able to update the database schema. If any user could update the schema in a database transaction, all subsequent transactions that try to access the updated table would fail, and the user might be confused why the transaction failed. The database administrator must lock the system, update the database schema from his own log in interface, and unlock Bridge-DB again to allow users to send queries.

5. IMPLEMENTATION

5.1 Translation Process

As mentioned in Section 4.2, the translation from BQL to Cypher or SQL happens after the TranslationController has translated the JSON object into our own Query object. The Query object is then sent to each translator to be translated to their respective language. The translators loop through each query in the queue and look at the `type` of the queries. Depending on the type of query the translators call the corresponding `generate()` function. For example, if the query type is `insert`, the translators call `generateInsert()`. Each translator has its own `generateInsert()` function specific to that translators language. The translated queries are then added to a running query array inside the translators. All queries in the received Query object are translated before returning the query array to the TranslationController.

The translation implementation of different query types is very similar to each other. First all relevant information is extracted from the query object, e.g. the table name, column names and values. The queries are then generated based on the rules of the external query languages.

Listing 8: Example of translation from Query object to Cypher

```
1 public function generateRenameTable($query) {
2   $table = $query['table'];
3   $new   = $query['new'];
4
5   $cypherQuery = "MATCH(n:".$table.") REMOVE n:
        ".$table." SET n:".$new.";";
6
7   return $cypherQuery;
8 }
```

Listing 8 shows the Cypher translation of a query for renaming a table. As can be seen on Line 2 and 3, the old and new table name is extracted from the query. Line 5 shows how the Cypher query is created. Lastly on Line 7 the Cypher query is returned, which is then added to the query array. This translation is very basic. Other translations involve significantly more complicated language constructs because of for example primary keys, foreign keys, and unique values.A more complicated translation example is shown in Listing 14, which can be found in Appendix B. This example actively uses the IDS to get information about the table, such as the primary key, foreign keys, and counter, and modifies the translation process depending on which information is provided. For example, if the user does not provide a primary key in the insert query, then the translator uses the counter value as primary key. The existance of foreign keys also effect the translation processes, since in Neo4J we must find and connect nodes depending on the provided foreign keys.

5.2 Transaction Control

The TransactionController has replaced many of the main functions of the connectors in Bridge-DB v.1.0. The TransactionController connects to the external databases, and handles the commit and rollback of a transaction depending on the returned results from each database. The external databases are responsible for validating the queries send to them. If a query fails in one of the external databases then the TransactionController receives an error and responds by performing a rollback on the database that failed.

The TransactionController has three main parts. First a connection is established using the `addDatabase()` function in Listing 9. This connection is stored in the `clients[]` array using the database name as the key on Line 10. So if we want to get the Neo4J connection, it is done by calling `clients['neo4j']`. The same is true for all other databases. Also note, that the query array is stored in the same way in the `queues[]` array on Line 11. The `failed[]` array tells if the transaction failed or committed. The `message` variable stores the message returned to the user. If the transaction was committed, the TransactionController will return a 'Transaction Committed' message. Otherwise an error message is returned.

Fault tolerance is implemented in order to support the eventual consistency property. Additionally the basically available property is supported through the implementation of both the heuristic optimizer and the TransactionController, since Bridge-DB v.2.0 allows the users to read from databases even if they are locked as long as a connection is established. Whether a database is locked or unlocked is indicated by the existence of a lock file as illustrated on Line 5. The TransactionController checks on Line 9 whether the connection is established, the database is locked, and whether the query is a select query. If this is the case the TransactionController proceeds as it normally would. If the connection to the database can not be established or if the database is locked, then the transaction is pushed to a failed transaction file on disk, and the database will be locked by creating a lock file if one did not already exist. This is handled by the `push_and_lock` function seen in Listing 12. The CRON job is responsible for unlocking the database again. This is described in Section 5.3.

After the established connections and query arrays have been stored in the `clients[]` and `queues[]` arrays respectively, the transaction start to execute its queries on the external databases. This is done by the `prepareTransaction()` function seen in Listing 10. On Line 5 a transaction is started for PostgreSQL using the `pg_query` function with the PostgreSQL connection and `BEGIN;` as input. `pg_query` sends the provided string to be executed on the provided connection. On Line 7-23 we try to execute all SQL queries of the transaction. First we get the status of the PostgreSQL connection. If the connection was lost, the transaction is handled by the `push_and_lock` function. If the connection is `OK` Line 11, then the query is executed on PostgreSQL and the result is stored in `queryResult`. PostgreSQL returns false if it was unable to execute the query. If the query failed, then `failed['postgresql']` is set to true on Line 15 and the loop breaks on Line 16. Note, this transaction will not be stored in a database specific FTF. The same is true if an unexpected exception was thrown on Line 31-36. At this point PostgreSQL has only tried to execute each query in the transaction, but nothing has been committed yet. This is handled by the function `commit_or_rollback()` shown in Listing 11. The prepareTransaction function is the same for Neo4J.

Listing 9: addDatabase function

```
1  private function addDatabase($db, $json) {
2    $data = json_decode($json);
3    switch ($db) {
4      case 'postgresql':
5        $isLocked = file_exists("lib/locks/".$db.
             ".lock");
6        $conn = pg_connect(" host=".$data->host."
              dbname=".$data->name." user=".$data
             ->user." password=".$data->password."
             ") or die('Connection Failed');
7  // FAULT TOLERANCE - MISSING CONNECTION
8  // IF CONNECTION IS ESTABLISHED, BUT THE
       DATABASE IS LOCKED AND THE QUERY IS A READ
       QUERY THEN PROCEED ANYWAY
9        if($conn !== false && $isLocked && $this
             ->isSelect($data->queue[0])) {
10         $this->clients[$db] = $conn;
11         $this->queues[$db] = $data->queue;
12         $this->failed[$db] = false;
13       }
14 // IF DATABASE IS LOCKED OR CONNECTION COULD
       NOT BE ESTABLISHED THEN PUSH FAILED
       TRANSACTION
15       else if($isLocked || $conn === false) {
16         $this->message = $conn;
17         $this->push_and_lock($db, $data->queue)
               ;
18         $this->failed[$db] = true;
19       }
20       else {
21         $this->clients[$db] = $conn;
22         $this->queues[$db] = $data->queue;
23         $this->failed[$db] = false;
24       }
25       break;
26       [...] // Similar for Neo4J
27     default:
28       break;
29   }
30 }
```

If no failure occurred, the transaction is committed on Line 3 and 15 in Listing 11. Otherwise the transaction is rolled back on Line 8 and 18. Normally the Neo4J transaction will automatically rollback if an error occurred while executing Cypher queries in the transaction, but just in case it did not, we do it manually on Line 6-8. If the transaction was successful, the result is returned to the user. The next section will discuss what happens in case of failure. This will also explain how the CRON job unlocks a database.

5.3 Implementing Fault Tolerance and BASE

We wanted to implement a weak consistency model, that would allow one database to be available even if a connection to the other could not be established. We looked at three failures that our model should take into account. This section shows how we have implemented fault tolerance into

Listing 10: prepareTransactions function

```
1  private function prepareTransactions() {
2    foreach ($this->queues as $db => $queue) {
3      switch ($db) {
4        case 'postgresql':
5          pg_query($this->clients[$db], "BEGIN; ");
6          foreach ($queue as $sql) {
7            try {
8              $stat = pg_connection_status($this->clients[$db]);
9  // FAULT TOLERANCE - CONNECTION LOST WHILE ADDING QUERY TO TRANSACTION
10 // IF CONNECTION IS OK THEN TRY TO ADD THE NEXT SQL QUERY TO THE TRANSACTION
11             if($stat === PGSQL_CONNECTION_OK) {
12               $queryResult = pg_query($this->clients[$db], $sql);
13               if($queryResult === FALSE) {
14                 $this->message[] = pg_last_error($this->clients[$db]);
15                 $this->failed[$db] = true;
16                 break;
17               }
18               else {
19                 while($row = pg_fetch_row($queryResult)) {
20                   $this->result[$db][] = $row;
21                 }
22               }
23             }
24 // IF CONNECTION IS NOT OK THEN PUSH FAILED TRANSACTION AND LOCK DATABASE
25             else {
26               $this->message = "Error: ...";
27               $this->push_and_lock($db, $queue);
28               break;
29             }
30           }
31           catch(Exception $e) {
32 // FAULT TOLERANCE - ADDING SQL QUERY FAILED
33             $this->message = pg_last_error($this->clients[$db]);
34             $this->failed[$db] = true;
35             break;
36           }
37         }
38         break;
39         [...] // Similar for Neo4J
40       default:
41         break;
42     }
43   }
44 }
```

Listing 11: Commit or rollback function

```
1  private function commit_or_rollback() {
2    if($this->clients['neo4j'] && $this->failed['neo4j'] === false) {
3      $this->transactions['neo4j']->commit();
4    }
5    else {
6      if($this->clients['neo4j']){
7        if(!$this->transactions['neo4j']->isClosed()) {
8          $this->transactions['neo4j']->rollback();
9  // Make sure transaction is closed if not already closed
10       }
11     }
12   }
13
14   if($this->clients['postgresql'] && $this->failed['postgresql'] === false) {
15     pg_query($this->clients['postgresql'], "COMMIT;");
16   }
17   else {
18     pg_query($this->clients['postgresql'], "ROLLBACK;");
19   }
20
21   pg_close($this->clients['postgresql']);
22 }
```

Listing 12: Push and lock function

```
1  private function push_and_lock($db, $transaction) {
2    $queryString = $transaction[0];
3
4    if(strpos($queryString, "SELECT") === FALSE && strpos($queryString, " RETURN ") === FALSE){
5      _push_failed_transaction($db, $transaction); // Store failed transaction
6      $handle = fopen("lib/locks/".$db.".lock", 'w'); // Create lock file or open if it already exists
7      fclose($handle); // Close file
8      return "Failed transaction pushed to queue";
9    }
10   else{
11     return "Error: The Select Query did not go through. Please try again in 5 minutes or contact the database administrator for assitance";
12   }
13 }
```

Bridge-DB, and thereby implemented a weak consistency model that upholds the BASE properties.

The code that handles the fault tolerance is mainly implemented in the TransactionController. The implementation is illustrated in Listing 9 and Listing 10. The function `push_and_lock` in Listing 12 handles the locking of the database on Line 6 by trying to open the lock file. If the lock file does not exist it is created. Line 7 closes the file. On Line 5 `_push_failed_transaction()` pushes the failed transaction to a FTF.

Note, that only transactions that fail because of either a missing connection or a lost connection are stored in a FTF. Failed transactions are stored in a database specific file, meaning a failed PostgreSQL transaction is stored in the file `failed_postgresql` and vice versa for Neo4J.

Listing 13 on the following page shows the implementation of the CRON job, which is responsible for getting the next failed transaction from the FTF (Line 9) and passing

it to the FailedTranactionController (Line 25-30). If the FailedTransactionController informs the CRON job, that the transaction committed, then it is removed from the FTF on Line 37. If the transaction failed, then the CRON job will try again. Note, that if the transaction failed because for example PostgreSQL was unable to execute a query in the transaction because of an error in the query, then the transaction is also remove from the FTF. This is done to avoid endlessly trying to execute a transaction that can not be executed on a database.

On Line 9 where the CRON job gets the next failed transaction, if no failed transaction exist in the FTF, then the database is unlocked by deleting the lock file on Line 12. The regular TransactionController is then free to send new transactions to the unlocked database.

Listing 13: CRON job

```
1  function run($db_list){
2    $clients = array();
3    foreach ($db_list as $db => $values) {
4      if(file_exists("lib/locks/".$db.".lock")) {
5        switch ($db) {
6          case 'postgresql':
7            $conn = pg_connect(" host=".$values['
                 host']." dbname=".$values['name'
                 ]." user=".$values['user']."
                 password=".$values['password']);
8            if($conn !== FALSE) {
9              $sql_transaction =
                   _next_failed_transaction($db);
10             if($sql_transaction['transaction']
                   === false) {
11 // Unlock PostgreSQL if no more failed
       transactions exist
12               unlink("lib/locks/postgresql.lock
                     ");
13             }
14             else {
15               $clients[$db] = json_encode(array
                     ('queue'=>$sql_transaction['
                     transaction'], 'host'=>
                     $values['host'], 'name'=>
                     $values['name'], 'user'=>
                     $values['user'], 'password'=>
                     $values['password']));
16             }
17           }
18           break;
19           [...] // Similar for Neo4J
20         default:
21           break;
22       }
23     }
24   }
25   $api = new RestClient(array(
26     'base_url' => "http://127.0.0.1/trans/
           failed/",
27     'format' => 'json',
28   ));
29   // SEND TRANSACTION TO FAILED TRANSACTION
         CONTROLLER
30   $result = $api->post("/", $clients);
31   foreach ($clients as $db => $key) {
32     if($res['failed'][$db] === 1) {
33       //TRANSACTION FAILED AGAIN
34     }
35     else if($res['failed'][$db] === 0) {
36       // IF TRANSACTION WAS SUCCESSFULL THEN
             REMOVE FROM FAILED TRANSACTION QUEUE
37       _pop_failed_transaction($db);
38     }
39   }
40
41   return $res;
42 }
```

6. EXPERIMENTS

In this project we analyze the response time, the TransactionController performance, and fault tolerance.

The response time analysis shows the strengths and weaknesses of the external heterogeneous databases. This data can be used to improve the heuristic optimizer further.

We investigate whether the performance of the TransactionController depends on the queried external database. We do this by examining the difference between evaluating a query on PostgreSQL and Neo4J. The query will return a result set containing over 1 million strings and we will limit the amount of results displayed to the user. We will examine how the two databases perform in relation to each other and in relation to the amount of results returned to the user.

We must ensure that the BASE properties are supported by Bridge-DB v.2.0. Our fault tolerance test will go through the three failure cases described in Section 4.2.1 to show that Bridge-DB will recover correctly from any failure and all external databases eventually will become consistent. At the same time we will test that at least one external database is always available, thereby fulfilling the basically available property.

In this project we will not document, that the heuristic optimizer chooses the best external database, since we have only made changes that affect the availability property, which is tested during the fault tolerance test. We therefore refer to our previous work for experimental evidence, that the optimizer works [10].

6.1 Data set

We use the IMDB data set used in our previous work [10]. However, we are modified the Neo4J database to store the `Crew` table as nodes instead of edges. This was done in order to allow tables with more than two foreign keys as described in Section subsection 2.2.

The data set consists of three tables; `Person`, `Crew`, and `Title`.

For each tuple/node in the `Crew` table, Neo4J also contains two edges, one to a `Person` node and another to a `Title` node. This is done to illustrate the foreign key relation of the PostgreSQL database. The size of the Neo4J database

Table	Size
Person	1,135,538
Crew	5,063,372
Title	351,429

Table 2: Dataset Size

is 5.36 GB and 607 MB on PostgreSQL. We counted the amount of tuples and nodes in PostgreSQL and Neo4J after measuring the database size, and all tables contain the same amount data. However, PostgreSQL compresses the stored data and is therefore able to store the same amount of information on much less disk space [15].

6.2 Setup

The tests are performed on a single Windows 8.1 Lenovo laptop with an Intel Core i7-4710HQ CPU @ 2.50GHz and 8 GB RAM. The tests are run through XAMPP's (open source cross-platform web server solution stack package) Apache HTTP Server and executed on the Chrome browser version 43.0.2357.81.

6.2.1 Test Queries

The test queries can be found in Appendix C in their BQL, SQL, and Cypher format. Table 3 provides a short description of the 10 queries used in the Response time test as well as the result size. Note, that results can be returned in different formats. The ones used in our tests include strings, integers, and rows. A row in PostgreSQL corresponds to a node in Neo4J.

Query	Description	Result Size
1	Find all person names	1,135,538 strings
2	Count all crew tuples/nodes	1 string
3	Find Kevin Bacon	1 row/node
4	Find all movie titles staring Will Smith	71 strings
5	Find all distinct movie title of Will Smith's co-actors	19,811 strings
6	Find all distinct co-actors of Will Smith	3,137 rows/nodes
7	Find all movie titles	351,429 strings
8	Find all movie titles produced before 2013	200,967 strings
9	Find all movie titles after 2013	150,462 strings
10	Find the amount of titles separating Will Smith and Kevin Bacon	1 integer

Table 3: Query Set

6.3 Response Time Test

In the first test we run the 10 queries in Table 3 on PostgreSQL and Neo4J. Query 1-10 will be executed on PostgreSQL on a warm cache. Afterwards the experiment is repeated on Neo4J. We repeat each query 10 times and find the average executing time by taking the sum of all results, removing the highest and lowest result, and dividing by 8. We manually control which database the query should be executed on. This means, that the heuristic optimizer is temporarily disabled in the translators. To see a comparison between cold and warm cache performance we refer to our previous work [10]. During the test we measured times on three different places; pre-processing, evaluation, and post-processing. Pre-processing includes converting JSON to the Query object and translating the Query object to both SQL and Cypher. Evaluation time is the time used by either PostgreSQL or Neo4J to evaluated the query, and post-processing includes processing 1000 of the results to be displayed and the communication between the TranslationController and the TransactionController. It should be noted that PostgreSQL and Neo4J return the same amount of results for each query, but we decide how many of these results are displayed to the user. The browser interface of Neo4J has a limit of 1000 displayed nodes, which is why we have set the display limit to 1000. When taking the total response time and subtracting the pre-processing, evaluation, and post-processing times, we get a rest time. This rest includes the communication time between the user and the TranslationController and is also included in the results.

6.3.1 Results

Table 4 and Figure 6 show the average response time for both PostgreSQL and Neo4J for query 1-10. The complete result set can be found in Appendix E.

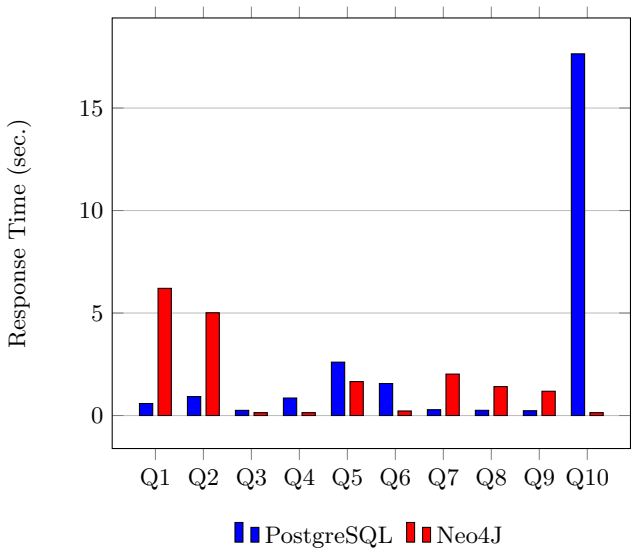

Figure 6: Query response time comparison (PostgreSQL/Neo4J)

The pre-processing time did not change significantly from each query. We therefore calculated the average pre-processing time using the same method as before, e.i. removing the highest and lowest test results and taking the average of the remaining results, and found that Bridge-DB v.2.0 uses 6.4 milliseconds in the pre-processing phase.

Post-processing varied a little and was slightly different between PostgreSQL and Neo4J. Bridge-DB used a minimum of 78.4 milliseconds and a maximum of 119 milliseconds for to process PostgreSQL results, while Neo4J results are processed in between 89.8 and 517.3 milliseconds. The highest processing time used for Neo4J was in query 1, where the number of returned results exceeds 1 million strings. Bridge-DB processed the same amount of PostgreSQL results in 95.1 milliseconds. Processing Neo4J results are clearly more dependent on the amount of results to process than PostgreSQL. PHP includes specialized functions to process PostgreSQL results, which might explain the increased processing performance, since Neo4J results are processed using the Neo4J library 'Everyman' [3].

	PostgreSQL					Neo4J				
Query	**Response**	**Pre**	**Evaluation**	**Post**	**Rest**	**Response**	**Pre**	**Evaluation**	**Post**	**Rest**
1	0.586503	0.005827	0.447168	0.095073	0.032874	6.204515	0.007010	5.633367	0.517285	0.035397
2	0.922511	0.007399	0.792883	0.083348	0.039304	5.013502	0.006902	4.871068	0.100673	0.033245
3	0.252922	0.005922	0.133931	0.078419	0.033014	0.141530	0.006630	0.007250	0.091183	0.035835
4	0.853740	0.005361	0.730760	0.082626	0.034786	0.140595	0.006679	0.007183	0.091629	0.037010
5	2.603812	0.005530	2.464579	0.094696	0.034541	1.654690	0.006992	1.477047	0.124712	0.040660
6	1.560230	0.005916	1.396845	0.118954	0.038502	0.222248	0.006875	0.065982	0.112240	0.033615
7	0.284467	0.006730	0.147722	0.090885	0.035837	2.021189	0.008411	1.747961	0.230280	0.033902
8	0.254271	0.005832	0.114373	0.089137	0.042594	1.410634	0.006387	1.199953	0.172674	0.030962
9	0.233845	0.006348	0.098002	0.089503	0.038730	1.184996	0.006642	0.990444	0.158594	0.033256
10	17.643363	0.005719	17.518356	0.082096	0.039378	0.140083	0.006272	0.017549	0.089819	0.027227

Table 4: Average results measured in seconds for each phase.

The remaining time used to transport results back to the user from the TranslationController was consistent for both PostgreSQL and Neo4J. We calculated the average communication time to be 3.5 milliseconds.

6.3.2 Discussion

From Figure 6 we can see that query 1, 2, and 7-9 execute faster on PostgreSQL than Neo4J. Especially query 1, which finds all the names in the database. This query returns over 1 million string results, but is significantly faster on PostgreSQL since it only needs to look up the names in the `Person` table. Neo4J must go through all nodes in the database, find the nodes labeled `Person` and return the name. The same is true for query 7, 8, and 9, where we find all movie titles, however, the amount of tuples/nodes is much smaller (351,429 strings) in the `Title` table. Query 8 and 9 also finds all movie titles but limits the result by production year, before or after 2013 respectively.

Query 4-6 perform better on Neo4J than PostgreSQL by between 0.7 and 1.3 seconds. These queries all involve more than 1 join operation on PostgreSQL. Query 4 finds all the movie titles staring Will Smith. This query involves 2 join operations, one between the `Person` and `Crew` tables, and one between the `Crew` and `Title` tables. Query 5 finds all the movie titles starting a co-actor of Will Smith. This query involves 6 join operations, 4 to find all co-actors of Will Smith (as done in Query 6) and 2 additional to find the movie titles staring these co-actors. Note, that co-actors is a broad term in this database. This also involves all the people involved in a movie, such as writers, directors, and producers. From these results it is becoming clear, that if a query involve more than 2 join operations on a significantly large enough dataset, then Neo4J outperforms PostgreSQL.

The most noticeable difference is in query 10, where we test the special reachability graph query. Neo4J is much faster in these situations. The PostgreSQL query 10 is a recursive query, which involves a large amount fo join operations. Reachability is a common known graph problem and efficient algorithms have been designed to solve the problem. Neo4J utilize these algorithms to prove the same result as PostgreSQL much faster, since Neo4J avoids the large amount of join operations necessary in PostgreSQL.

6.4 TransactionController Performance Test

We run this test to examine whether Bridge-DB processes PostgreSQL results differently than Neo4J results as well as analyze how much time was used on transporting the results back from the TransactionController to the TranslationController. This was done by changing the display limit. This test was conducted because of the response time test results, which indicates that Neo4J results are processed slower than PostgreSQL results.

The display limit of 1000 was set based on the limit set by the Neo4J browser interface. However, we wanted to analyze the effect on the post-processing time, if the amount of displayed results was increased. In this test we set the display limit at 10, 100, 1 thousand, 10 thousand, 100 thousand, and 1 million. We executed query 1 on each database 10 times as in the response time test and calculated the average using the same formula. Query 1 is used, since it returns 1,135,538 strings as a result.

In our analysis, we do not want the evaluation time of the external databases to affect the result, since we only want to see the effect on the post-processing time. We therefore subtract the evaluation time from the post-processing time.

We measure three times during this test. The TransactionController response time, i.e. the time from the moment the TranslationController sends the query array to the TransactionController until the TransactionController returns a result. The TransactionController processing time, i.e. the time spend in the TransactionController minus the evaluation time used by the external database. We calculate the transport time, i.e. the time it takes to get data between the TransactionController and the TranslationController by subtracting the TransactionController processing time from the TransactionController response time.

6.4.1 Results

Table 5 and Table 6 display the average processing analysis results for PostgreSQL and Neo4J respectively. The results are split into the three measured times as described above; response time, transport time, and processing time. The time results are shown in seconds. Figure 7 and Figure 8 help to illustrate the difference between PostgreSQL and Neo4J, and how they react to an increased amount of results to process. Note, that both figures use the logarithmic scale for both the x and y axis. It is also important to mention that both PostgreSQL and Neo4J return the same amount of results. We only limit the amount of results returned back to the user. The total result set can be found in Appendix F.

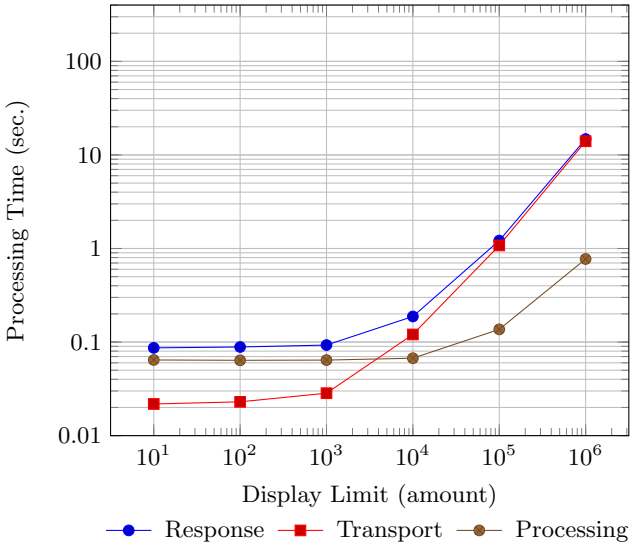

Figure 7: PostgreSQL Processing Analysis Chart

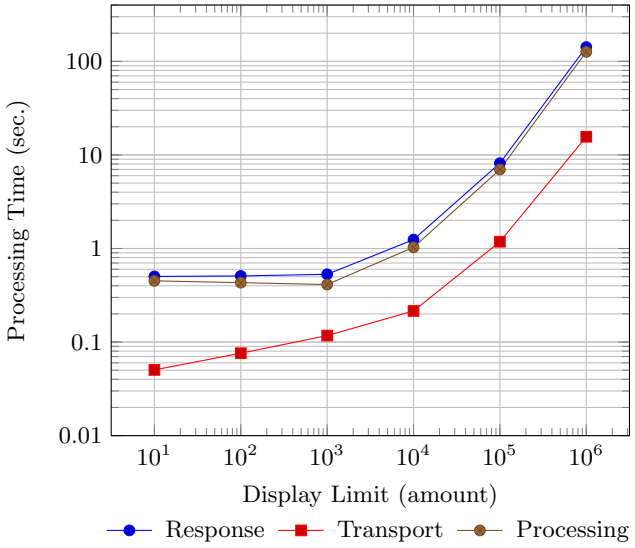

Figure 8: Neo4J Processing Analysis Chart

Display limit	Response	Transport	Processing
10	0.0866757	0.0217925	0.0643718
100	0.0886086	0.0229726	0.0638434
1,000	0.0928630	0.0284130	0.0641874
10,000	0.1877267	0.1206022	0.0672238
100,000	1.2146479	1.0774697	0.1366803
1,000,000	14.7836617	14.0070932	0.7723322

Table 5: PostgreSQL Processing Analysis Results

Display limit	Response	Transport	Processing
10	0.502750	0.050425	0.451193
100	0.509014	0.075984	0.431441
1,000	0.530100	0.117238	0.412069
10,000	1.244197	0.215093	1.030081
100,000	8.160219	1.181496	6.981745
1,000,000	142.233858	15.643400	126.612218

Table 6: Neo4J Processing Analysis Results

6.4.2 Discussion

Our experiment returned some interesting results, since, as seen in Figure 7 and Figure 8, PostgreSQL and Neo4J results are processed very differently.

Until the display limit reaches over 10 thousand, PostgreSQL results are processed in under 100 milliseconds and even 1 million displayed results are processed in under 1 second. The transportation time is increased from close to 20 milliseconds to a little over 14 seconds, which is very similar to the transportation results for Neo4J.

However, Neo4J results are processed significantly slower. Processing time is never below 100 milliseconds. We see from Table 6 that it makes no difference whether Bridge-DB processes 10 or 1000 results. Even though the average results shows that 1000 results are processed faster than 10, the difference is so small that it lies within the margin of error. 10 thousand results are processed in just about 1 second, and in 6 seconds for 100 thousand results. The most surprising result is that it takes Bridge-DB more than 126 seconds to process 1 million Neo4J results in comparison to only 770 milliseconds for PostgreSQL. We suspect that the Neo4J library used to process the result might be a bottleneck.We can conclude that the processing time is a big factor for especially Neo4J. This means that in some cases a query that would evaluate faster on Neo4J would be processed slow enough, that PostgreSQL would still be the best database to use. A good future cost model would need to take this extra processing time into consideration.

6.5 Fault Tolerance

Our implementation of fault tolerance into Bridge-DB ensures that the BASE properties are upheld. It ensures that one database is available even if another is locked, and that the all other databases will eventually be consistent. The purpose of this test is to prove this claim. The transactions used in this test can be found in Appendix D.

We test the three failure cases discussed in Section 4.2.1, namely missing connection, connection loss, and error in a transaction. All three tests consist of the same three main transactions. The first transaction inserts a new movie into the database, the second updates the title, and lastly the third deletes the movie. Before each test is performed we make sure that both external databases do not already contain the new movie. Between each transaction we perform a select on both external databases to make sure that the transaction committed. Note, that we also take this opportunity to make sure, that our heuristic optimizer selects the available external database even if the query would normally be executed faster on the unavailable database. We use time marks to indicate when a transaction or event occurs. T1 is the first time mark followed by T2 etc.

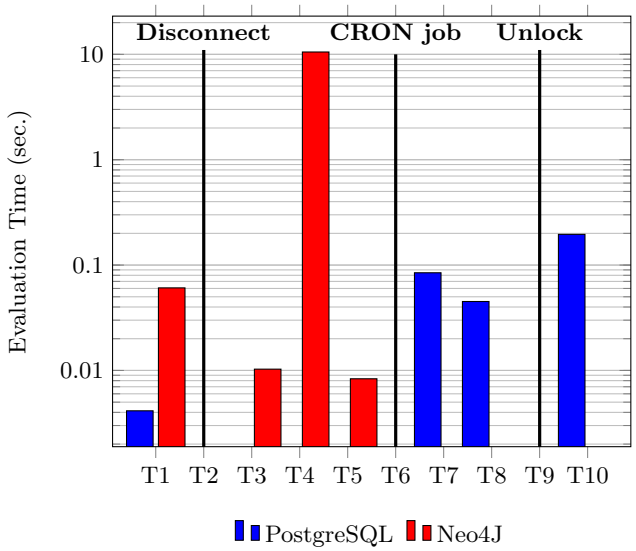

Figure 9: Missing Connection Chart - disconnect PostgreSQL

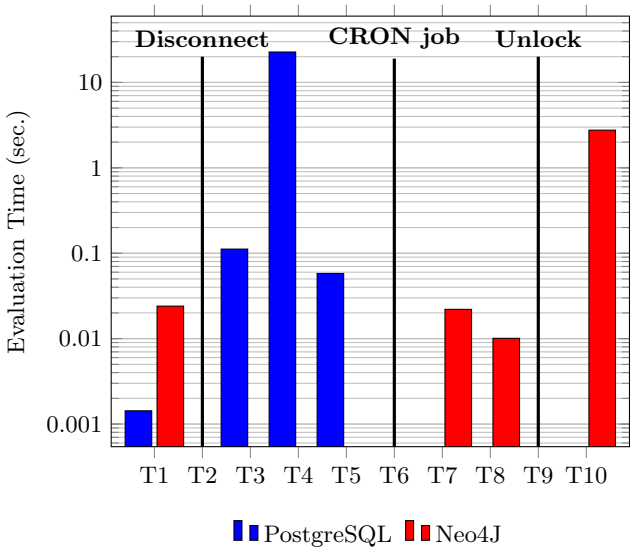

Figure 10: Missing Connection Chart - disconnect Neo4J

6.5.1 *Missing Connection*

The following list shows the steps of the missing connection test:

- T1 Insert new movie named 'Avengers: Civil War' into all databases
- T2 Disconnect PostgreSQL
- T3 Update title of movie from 'Avengers: Civil War' to 'Captain America: Civil War'
- T4 Execute Query 7
- T5 Delete new movie using 'Captain America: Civil War' as the WHERE-condition
- T6 Reconnect PostgreSQL and start CRON job
- T7 CRON job executes update transaction
- T8 CRON job executes delete transaction
- T9 CRON job stops and unlocks PostgreSQL
- T10 Execute Query 7

We measure the evaluation time of the external databases and use these times to show, that they either executed a transaction or did not. Note, that at T4 when we execute query 7, the heuristic optimizer would regularly perform this query on PostgreSQL. However, since PostgreSQL is disconnected, the optimizer should choose to execute the query on Neo4J. The optimizer should choose to execute query 7 on PostgreSQL at T10 after PostgreSQL is unlocked at T9.

The test is repeated where we cut the connection to Neo4J instead of PostgreSQL. We also use query 10 at T4 and T10 instead of query 7, since query 10 would normally be executed on Neo4J instead of PostgreSQL, since it involves graph reachability.

Results and Discussion

Figure 9 and Figure 10 shows the evaluation time of the external databases at different time marks. At T1 both databases are active and insert the new movie as expected. We then disconnect PostgreSQL or Neo4J at T2. Depending on which database is disconnected we see, that the update transaction is only performed on one database. Figure 9 shows that at T4 the heuristic optimizer chooses to execute query 7 on Neo4J since PostgreSQL is disconnected. After PostgreSQL is unlocked at T9, we see at T10 that query 7 is executed on PostgreSQL and that the evaluation time is much faster than the evaluation time on Neo4J. The same results can be seen in Figure 10 where PostgreSQL executes query 10 at T4, but Neo4J takes over at T10 after being unlocked at T9.

Another result is that the CRON job executes the update and delete transactions in the correct order at T7 and T8.

6.5.2 *Connection Loss*

The previous test is repeated, however, the separate update and delete transactions are combined into one transaction. This allows us to disconnect PostgreSQL between the update and delete query, thereby simulating a connection loss.

The following list shows the steps of the lost connection test:

- T1 Insert new movie named 'Avengers: Civil War' into all database
- T2 Execute update and delete as one transaction, but cut connection to PostgreSQL after the update to simulate connection loss
- T3 PostgreSQL is locked
- T4 Execute Query 7
- T5 Reconnect PostgreSQL and start CRON job

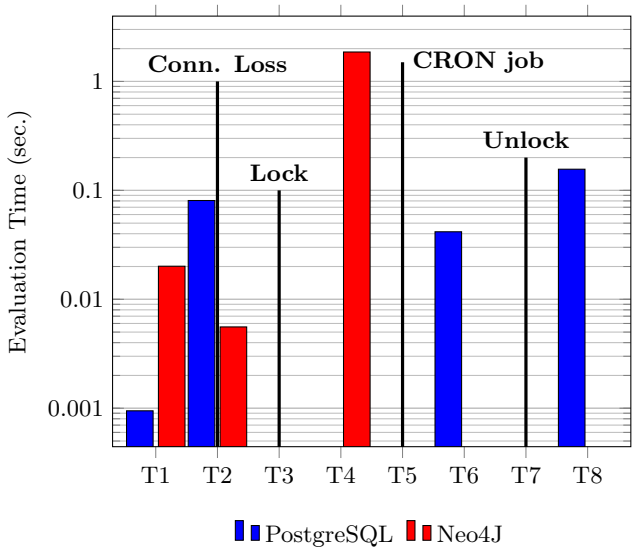

Figure 11: Lost Connection Chart - lose PostgreSQL connection after update query

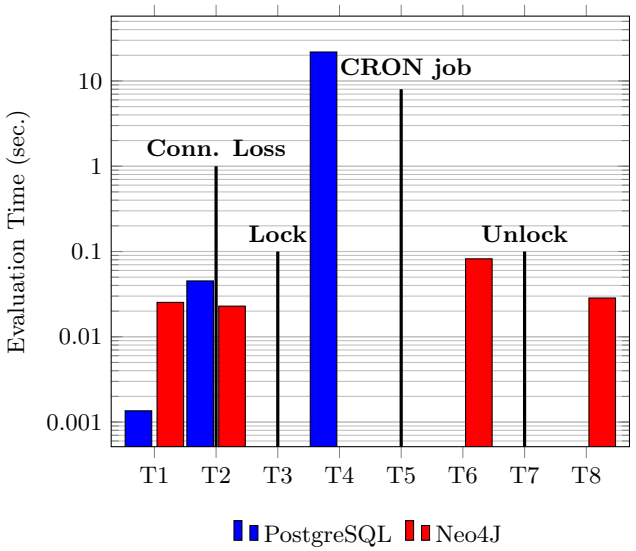

Figure 12: Lost Connection Chart - lose Neo4j connection after update query

T6 CRON job execute the update and delete transaction

T7 CRON job stops and unlocks PostgreSQL

T8 Execute Query 7

The test is repeated where we disconnect Neo4J instead of PostgreSQL and use query 10 at T4 and T8 instead of query 7.

Results and Discussion

The connection loss tests illustrated in Figure 11 and Figure 12 shows the same results as the missing connection test. The heuristic optimizer chooses to execute query 7 or 10 at T4 on the only available database. The CRON job commits the combined update and delete transaction at T6 in a single step and unlocks the locked database such that either query 7 or 10 can be executed on the most compatible database.

6.5.3 Transaction Failure

A transaction can fail because of for example syntax errors or invalid queries that try to insert using an existing primary key. Whether it is a syntax error or invalid query, the transaction failure is the same. In order to test the case where a transaction fails, we manually force an error to occur in the update transaction by editing the `generateUpdate()` function such that it will case a syntax error.

The following list shows the steps of the transaction failure test:

T1 Insert new movie named 'Avengers: Civil War' into all database

T2 Perform failed update transaction and get error message

T3 Execute Query 7

T4 Execute Query 10

T5 Delete movie using 'Captain America: Civil War' title as WHERE-condition

Note, that the new movie will not be deleted from the database, since the update function failed on both databases. PostgreSQL is not locked at T3, so Query 7 should be executed on PostgreSQL. Neo4J should similarly execute query 10 at T4.

6.5.4 Results and Discussion

Figure 13 shows that Bridge-DB is not affected by failed transactions. Only PostgreSQL use a small amount of time to process the failed update transaction at T2. The results also show that the heuristic optimizer works, since it evaluates query 7 on PostgreSQL at T3 and query 10 on Neo4J at T4. At T5 we try to delete a movie named 'Captain America: Civil War', but since the update failed, it is not deleted. Neo4J is faster at rejecting the delete transaction at T5, however, both PostgreSQL and Neo4J execute in under 100 milliseconds, which we find to be a good result.

6.5.5 Summary

The results confirm that Bridge-DB v.2.0 upholds the BASE properties. In the first two tests at T4 Bridge-DB proves to be basically available by executing a query on a less compatible but available database. Likewise, the eventual consistency property is upheld at T7 and T8 by the CRON job in the first test. The same is true at T6 for the connection loss test. Lastly, the soft state property can be said to be upheld, since the CRON job would normally not be started manually but would automatically run to recover failed transactions. This can be seen as action without input, which is one of the ways of interpreting the soft state property of BASE. The last test shows that failed transactions do not effect the external databases, if the cause of the failure is a syntax error. This ensures, that eventual translation errors will

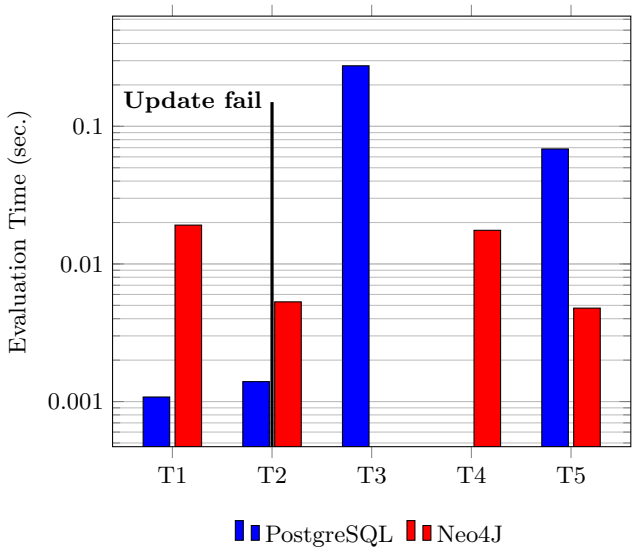

Figure 13: Failed Transaction Chart - update failed to execute on both Neo4J and PostgreSQL

not result in locked databases, which would affect all users. The error is instead reported back to the user, who can then report the translation problem to a Bridge-DB responsible administrator. This error reporting could in the future be automated to ensure an up-to-date translator.

7. CONCLUSIONS

In this project the goal was to extend a middleware layer system to support both the CRUD operations as well as the BASE properties. We have developed Bridge-DB v.2.0 based on our previous work. The tests performed in this project show that Bridge-DB v.2.0 upholds the BASE properties by using a combination of new features in Bridge-DB v.2.0, such as the failed transaction file, the CRON job, an improved heuristic optimizer and the new Transaction-Controller. The translation has improved significantly from Bridge-DB v.1.0 and users are able to add more functions to BQL, since Bridge-DB still is designed with a modular approach.

In our processing performance test we showed, that future work on developing a cost model for Bridge-DB should consider the time it takes to process results, since PostgreSQL and Neo4J results are currently processed differently. This is relevant in cases where Neo4J would normally execute faster, but the processing time is high enough, that the total response time of running the transaction through PostgreSQL would still be smaller. In the first test we show that Neo4J is better at working with specialized graph traversal queries, but also handle select queries containing join operations at the same speed or faster than PostgreSQL.

We have shown that our middleware layer solution is a reliable solution when working with multiple heterogeneous databases and that it is possible to implement a consistent and reliable weak consistency model into such a middleware layer.

8. FUTURE WORK

8.0.6 *Distributed Bridge-DB*

Bridge-DB v.2.0 is a centralized middleware system, meaning all users must go through the same Bridge-DB server. Bridge-DB is run on one server while the connected external databases can be distributed throughout the network on other servers. Making Bridge-DB distributed, would mean that users could access any available Bridge-DB server. This would increase the availability of Bridge-DB, such that if one Bridge-DB server disconnects from the network users connected to that server could be reconnected to another Bridge-DB server and resume work. Another scenario is that a Bridge-DB server loses connection to one or more external databases. Other Bridge-DB servers could still have connection to the database and all queries could be redirected through one of these servers. Note, that this is different than our current implementation of the BASE properties, because Bridge-DB remains centralized in Bridge-DB v.2.0.

8.0.7 *Cost model*

The current version of Bridge-DB uses a heuristic optimizer. A more performable solution would be to build a cost function that analyses the query, and based on the analysis the query is send to the most suitable database. Some of the parameters on which to analyze would be network strength, CPU time, memory usage, join operations and a lot other operations. When analyzed on these parameters an optimal execution plan for the queries is constructed.

8.0.8 *Graphical User Interface*

A graphical user interface where queries can be written and results shown is required to ease the use of the system. This would need to include log-in functionality to restrict access to some BQL functions that change the underlying structure of the database. The design could be inspired by phpMyAdmin.net.

9. REFERENCES

[1] J. A. L. A. P. Sheth. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys*, 22(3), 1990.

[2] D. J. Abadi. Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story. *Computer*, 45(2):37–42, 2012.

[3] J. Adell. Web site. `https://github.com/jadell/neo4jphp/tree/master/lib/Everyman/Neo4j`.

[4] E. Brewer. CAP Twelve Years Later: How the "Rules" Have Changed. *IEEE Computer Society*, 2012.

[5] E. A. Brewer. Towards Robust Distributed Systems (Invited Talk). In *Principles of Distributed Computing*, 2000.

[6] J. Browne. Brewer's CAP Theorem. Web site, 2009. `http://www.julianbrowne.com/article/viewer/brewers-cap-theorem`.

[7] S. Gilbert and N. Lynch. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services. *SIGACT News*, 33(2):51–59, 2002.

[8] J. Gray. The Transaction Concept: Virtues and Limitations (Invited Paper). In *Proceedings of the Seventh International Conference on Very Large Data Bases - Volume 7*, pages 144–154, 1981.

[9] T. Haerder and A. Reuter. *Readings in Database Systems*. The MIT Press, 1988.

[10] C. K. Hansen, L. Nielsen, N. Cokljat, and R. S. Ettrup. Bridge-DB: A Middleware Layer for Distributed Multi-Database Systems. AAU Student Report, 2014.

[11] D. Heimbigner and D. McLeod. A Federated Architecture for Information Management. *ACM Transactions on Office Information System*, 3(3):253–278, 1985.

[12] I. Informix. Web site. `http://www-01.ibm.com/software/data/informix/`.

[13] N. R. Manuel Rodríguiz-Martínez. Mocha: A Self-Extensible Database Middleware System for Distributed Data Sources. *SIGMOD Rec.*, 29:213–224, 2000.

[14] M. Paradies, W. Lehner, and C. Bornhövd. GRAPHITE: An Extensible Graph Traversal Framework for Relational Database Management Systems. *CoRR*, abs/1412.6477, 2015.

[15] PostgreSQL Documentation. Web. `http://www.postgresql.org/docs/9.3/static/storage-toast.html/`.

[16] D. Pritchett. BASE: An ACID Alternative. *Queue*, 6(3):48–55, May 2008.

[17] The Apache Software Foundation. Web. `https://hadoop.apache.org/`.

APPENDIX

A. BQL V.1.0 AVAILABLE FUNCTIONS

Name	Input	Return
select	@param: string || empty =>*	@return self
distinct	@param: bool	@return self
from	@param: string	@return self
join	@param: table:string, condition:string	@return self
where	@param: key:string, values:mixed	@return self
or_where	@param: key:string, values:mixed	@return self
where_in	@param: key:string, values:array	@return self
or_where_in	@param: key:string, values:array	@return self
limit	@param: value:int, offset:int	@return self
offset	@param: offset:int	@return self
count	@param: id:string	@return self
traverse	@param: elm1:string, elm2:string	@return self
reachable	@param: elm1:string, elm2:string	@return self
send		@return json:object

Table 7: List of supported functions in BQL v.1.0

B. CYPHER TRANSLATION EXAMPLE

Listing 14: Advanced Translation into Cypher Insert Query

```
1  public function generateInsert($query){
2    $table   = $query->table;
3    $columns = $query->columns;
4    $values  = $query->values;
5
6    $f_keys  = _get_f_keys($table); //GET F KEYS
         FROM IDS
7    $p_key   = _get_p_keys($table); //GET P KEYS
         FROM IDS
8    $counter = _get_counter($table, 'neo4j'); //
         GET COUNTER FROM IDS
9
10   $p_index = array_search($p_key, $columns);
11
12   if($p_index !== FALSE && ($values[$p_index] <
          $counter || $values[$p_index] > $counter
         +10)){
13     return 'Primary key must be larger than '.
           $counter;
14   }
15
16   $cypherQuery = "";
17
18   // IF TABLE HAS F_KEYS FIND NODES WITH
         CORRECT ID
19   if($f_keys){
20     $cypherQuery .= "MATCH ";
21     for($i=0;$i<count($f_keys);$i++){
22       $cypherQuery .= "(a".$i.":".
             _get_f_key_table($table, $f_keys[$i])
             .")";
```

```
      $cypherQuery .= ($i<count($f_keys)-1) ? ", " : " WHERE ";
    }
    for($i=0, $j=0;$i<count($f_keys);$j++){
      if(strcmp($columns[$j], $f_keys[$i])==0){
        $cypherQuery .= "a".$i.".id=".$values[$j];
        $cypherQuery .= ($i<count($f_keys)-1) ? " AND " : "";
        $i++;
      }
    }
  }

  //MERGE: Avoids inserting identical nodes
  $cypherQuery .= " MERGE (n:".$table." { ";
  if($p_index === FALSE){ // IF USER DID NOT PROVIDE P_KEY THEN USE IDS COUNTER + 1
    $cypherQuery .= $p_key." : ".++$counter.", ";
    _set_counter($table, 'neo4j', $counter);
  }
  else{
    _set_counter($table, 'neo4j', $values[$p_index]);
  }
  for($i=0;$i<count($columns);$i++){
    $cypherQuery .= $columns[$i]." : ";
    $cypherQuery .= (is_numeric($values[$i])) ? $values[$i] : "'".trim($values[$i],"'|\"| ")."'";
    $cypherQuery .= ($i<count($columns)-1) ? ", " : "})";
  }
  if($f_keys){ // IF TABLE HAS F_KEYS THEN INSERT EDGES
    for($i=0;$i<count($f_keys);$i++){
      $cypherQuery .= " MERGE (a".$i.")<-[r".$i.":".$table."]-(n) ";
    }
  }
  $cypherQuery .= ";";

  return $cypherQuery;
}
```

C. RESPONSE TEST QUERIES

Listing 15: Query 1

```
// BQL:
$transaction->select("name")->from("Person")
->send();
// Generated SQL:
SELECT name FROM Person;
// Generated Cypher:
MATCH (z:Person) RETURN z.name;
```

Listing 16: Query 2

```
// BQL:
$transaction->select("Crew.*")
->from("Crew")->count(true)->send();
// SQL:
SELECT count(Crew.*) FROM Crew;
// Cypher:
MATCH (z:Crew) RETURN count(z);
```

Listing 17: Query 3

```
// BQL:
$transaction->select("*")->from("Person")
->where('Person.name', "Bacon, Kevin")->send();
// SQL:
SELECT * FROM Person
WHERE Person.name= 'Bacon, Kevin';
// Cypher:
MATCH (z:Person)
WHERE z.name='Bacon, Kevin'
RETURN z;
```

Listing 18: Query 4

```
// BQL:
$transaction->select("Title.title")
->from('Person')
->join('Crew', 'Crew.person_id=Person.id')
->join('Title', 'Crew.movie_id=Title.id')
->where('Person.name =', 'Smith, Will')->send()
    ;
// SQL:
SELECT Title.title FROM Person
JOIN Crew ON (Crew.person_id=Person.id)
JOIN Title ON (Crew.movie_id=Title.id)
WHERE Person.name = 'Smith, Will';
// Cypher
MATCH (z:Person)--(y:Crew)--(x:Title)
WHERE z.name ='Smith, Will'
RETURN x.title;
```

Listing 19: Query 5

```
// BQL:
$transaction->select("t3.title")
->distinct(true)->from('Person as p1')
->join('Crew as c1', 'c1.person_id=p1.id')
->join('Title as t1', 'c1.movie_id=t1.id')
->join('Crew as c2', 't1.id=c2.movie_id')
->join('Person as p2', 'p2.id=c2.person_id')
->join('Crew as c3', 'c3.person_id=p2.id')
->join('Title as t3', 't3.id=c3.movie_id')
->where('p1.name =', 'Smith, Will')->send();
// SQL:
SELECT DISTINCT t3.title FROM Person as p1
JOIN Crew as c1 ON (c1.person_id=p1.id)
JOIN Title as t1 ON (c1.movie_id=t1.id)
JOIN Crew as c2 ON (t1.id=c2.movie_id)
JOIN Person as p2 ON (p2.id=c2.person_id)
JOIN Crew as c3 ON (c3.person_id=p2.id)
JOIN Title as t3 ON (t3.id=c3.movie_id)
WHERE p1.name = 'Smith, Will';
// Cypher:
MATCH (z:Person)--(:Crew)--(:Title)--(:Crew)
    --(:Person)--(:Crew)--(t:Title)
WHERE z.name ='Smith, Will'
RETURN DISTINCT t.title;
```

Listing 20: Query 6

```
// BQL:
$transaction->select("p2")->distinct(true)
->from('Person as p1')->join('Crew as c1', 'c1.person_id=p1.id')
->join('Title as t1', 'c1.movie_id=t1.id')
->join('Crew as c2', 't1.id=c2.movie_id')
->join('Person as p2', 'p2.id=c2.person_id')
->where('p1.name =', 'Smith, Will')->send();
// SQL:
SELECT DISTINCT p2 FROM Person as p1
JOIN Crew as c1 ON (c1.person_id=p1.id)
JOIN Title as t1 ON (c1.movie_id=t1.id)
JOIN Crew as c2 ON (t1.id=c2.movie_id)
JOIN Person as p2 ON (p2.id=c2.person_id)
WHERE p1.name = 'Smith, Will';
// Cypher:
MATCH (z:Person)--(:Crew)--(:Title)--(:Crew)--(v:Person)
WHERE z.name ='Smith, Will'
RETURN DISTINCT v.name;
```

Listing 21: Query 7

```
// BQL:
$transaction->select("title")->from("Title")
->send();
// SQL:
SELECT title FROM Title;
// Cypher:
MATCH (z:Title) RETURN z.title;
```

Listing 22: Query 8

```
// BQL:
$transaction->select("title")->from("Title")
->where('production_year <=', '2013')->send();
// SQL:
SELECT title FROM Title
WHERE production_year <= 2013;
// Cypher:
MATCH (z:Title)
WHERE z.production_year <=2013
RETURN z.title;
```

Listing 23: Query 9

```
// BQL:
$transaction->select("title")->from("Title")
->where('production_year >', '2013')->send();
// SQL:
SELECT title FROM Title
WHERE production_year > 2013;
// Cypher:
MATCH (z:Title)
WHERE z.production_year > 2013 RETURN z.title;
```

Listing 24: Query 10

```
// BQL:
$transaction->reachable('Person AS n','Person AS m')->where('n.id', '1721442')->where('m.id', '92150')->send();
// SQL:
WITH RECURSIVE linkBetweenActors(idActor1, idActor2, degree)
AS (SELECT 92150, 92150, 0 UNION DISTINCT SELET DISTINCT link.idActor1, ai2.person_id, link.degree+1 FROM linkBetweenActors link
JOIN crew ai ON link.idActor2 = ai.person_id
JOIN crew ai2 ON ai.movie_id = ai2.movie_id
WHERE link.degree <3)
   SELECT degree
   FROM linkBetweenActors
   WHERE idActor1=92150 AND idActor2=395709
   LIMIT 1;
// Cypher:
MATCH ( n : Person ), ( m : Person ), p = shortestPath (( n ) -[*..15] - ( m )) WHERE n.id=1721442 AND m.id=92150 RETURN  length ([m in nodes(p) WHERE m:Title]) AS BaconNumber;
```

D. FAULT TOLERANCE TEST TRANSACTIONS

Listing 25: Insert Transaction

```
// BQL:
$transaction->insert("Title", ["title", "production_year"], ["Avengers: Civil War", 2016])->send();
// SQL:
INSERT INTO Title(id, title, production_year)
VALUES(3083395, 'Avengers: Civil War', 2016);
// Cypher:
MERGE (n:Title { id : 3083395, title : 'Avengers: Civil War', production_year : 2016});
```

Listing 26: Update Transaction

```
// BQL:
$transaction->update("Title", ['title'], ['Captain America: Civil War'], "title = Avengers: Civil War AND production_year = 2016")->send();
// SQL:
UPDATE Title
SET title = 'Captain America: Civil War'
WHERE title='Avengers: Civil War' AND production_year=2016;
// Cypher:
MATCH (n:Title)
WHERE n.title='Avengers: Civil War' AND n.production_year=2016
SET n.title='Captain America: Civil War';
```

Listing 27: Delete Transaction

```
// BQL:
$transaction->delete("Title", "title = Captain America: Civil War AND production_year = 2016")->send();
// SQL:
DELETE FROM Title
WHERE title='Captain America: Civil War' AND production_year=2016;
// Cypher:
MATCH (n:Title)
WHERE n.title='Captain America: Civil War' AND n.production_year=2016
OPTIONAL MATCH (n)<-[a]-(m)-[b]->()
WHERE str(type(a))=str(type(b))=str(head(labels(m)))
DELETE n, m, a, b;
```

E. POSTGRESQL AND NEO4J RESULT SET: QUERY 1-10

PostgreSQL

Query 1

Run	Response	Pre	Evaluation	Post	Rest
1	0.57808	0.00458	0.44771	0.08853	0.03727
2	0.60685	0.00622	0.44923	0.09600	0.05541
3	0.63552	0.00340	0.51916	0.09312	0.01984
4	0.57546	0.00489	0.43641	0.09831	0.03585
5	0.57015	0.00480	0.43556	0.10915	0.02064
6	0.59656	0.00552	0.47011	0.09418	0.02675
7	0.57175	0.00940	0.44321	0.08520	0.03395
8	0.61286	0.00647	0.44519	0.12218	0.03902
9	0.57552	0.00631	0.44791	0.08868	0.03263
10	0.57494	0.00784	0.43758	0.09262	0.03690
	0.58650	**0.00583**	**0.44717**	**0.09507**	**0.03287**

Query 2

Run	Response	Pre	Evaluation	Post	Rest
1	0.93235	0.00971	0.78338	0.10299	0.03627
2	0.90412	0.00647	0.78952	0.07421	0.03392
3	0.92875	0.01003	0.81267	0.06975	0.03629
4	1.02430	0.00860	0.80662	0.15855	0.05054
5	0.94253	0.00656	0.79592	0.08483	0.05522
6	0.89924	0.00687	0.78197	0.07406	0.03634
7	0.92407	0.00703	0.78787	0.09068	0.03849
8	0.91200	0.00736	0.80380	0.07508	0.02577
9	0.90334	0.00620	0.78540	0.07412	0.03762
10	0.93292	0.00659	0.79055	0.09081	0.04497
	0.92251	**0.00740**	**0.79288**	**0.08335**	**0.03930**

Query 3

Run	Response	Pre	Evaluation	Post	Rest
1	0.24588	0.00551	0.12996	0.07256	0.03786
2	0.23584	0.00562	0.13342	0.06849	0.02831
3	0.25087	0.00714	0.13781	0.08271	0.02321
4	0.24386	0.00759	0.13084	0.07100	0.03444
5	0.22824	0.00507	0.13791	0.06293	0.02233
6	0.25600	0.00485	0.13390	0.08399	0.03325
7	0.27860	0.00592	0.13481	0.08691	0.05096
8	0.25537	0.00499	0.13263	0.08143	0.03631
9	0.26048	0.00554	0.13784	0.08027	0.03683
10	0.27508	0.00796	0.13020	0.10303	0.03390
	0.25292	**0.00592**	**0.13393**	**0.07842**	**0.03301**

PostgreSQL

Query 4

Run	Response	Pre	Evaluation	Post	Rest
1	0.83627	0.00492	0.72063	0.07433	0.03639
2	0.84896	0.00851	0.73211	0.07493	0.03342
3	0.84894	0.00453	0.73932	0.06958	0.03551
4	0.89552	0.00546	0.75353	0.08822	0.04831
5	0.88567	0.00555	0.73820	0.08996	0.05196
6	0.86484	0.00565	0.74531	0.07664	0.03724
7	0.84310	0.00423	0.71725	0.09988	0.02174
8	0.85542	0.00403	0.72621	0.10819	0.01699
9	0.82702	0.00625	0.72596	0.06375	0.03107
10	0.84671	0.00630	0.71833	0.08748	0.03460
	0.85374	**0.00536**	**0.73076**	**0.08263**	**0.03479**

Query 5

Run	Response	Pre	Evaluation	Post	Rest
1	2.56517	0.00352	2.45863	0.08094	0.02207
2	2.55839	0.00487	2.41743	0.10067	0.03541
3	2.75381	0.00895	2.60422	0.10357	0.03707
4	2.58150	0.00867	2.42664	0.09237	0.05383
5	2.61348	0.00350	2.49454	0.08898	0.02646
6	2.58046	0.00452	2.44528	0.09257	0.03809
7	2.79564	0.00412	2.67578	0.09026	0.02548
8	2.59960	0.00565	2.43841	0.10216	0.05338
9	2.57808	0.00393	2.43148	0.11727	0.02540
10	2.53139	0.00959	2.39977	0.08698	0.03503
	2.60381	**0.00553**	**2.46458**	**0.09470**	**0.03454**

Query 6

Run	Response	Pre	Evaluation	Post	Rest
1	1.54675	0.00425	1.37796	0.11950	0.04504
2	1.55338	0.00561	1.40073	0.10961	0.03743
3	1.58067	0.00514	1.42295	0.11427	0.03831
4	1.63276	0.00872	1.40689	0.17678	0.04037
5	1.53443	0.00374	1.39043	0.10647	0.03378
6	1.57151	0.00774	1.41202	0.11063	0.04112
7	1.56920	0.00548	1.38581	0.12898	0.04893
8	1.56326	0.00761	1.40157	0.11357	0.04050
9	1.53675	0.00608	1.38357	0.11563	0.03147
10	1.56031	0.00541	1.39374	0.13943	0.02174
	1.56023	**0.00592**	**1.39685**	**0.11895**	**0.03850**

PostgreSQL

Query 7

Run	Response	Pre	Evaluation	Post	Rest
1	0.30302	0.00843	0.14713	0.11233	0.03514
2	0.29402	0.01006	0.15268	0.09038	0.04089
3	0.27796	0.00350	0.14603	0.09241	0.03602
4	0.29143	0.00546	0.14557	0.08895	0.05145
5	0.30262	0.00821	0.14836	0.11570	0.03035
6	0.26618	0.00559	0.14555	0.08858	0.02646
7	0.29421	0.00869	0.14968	0.09005	0.04580
8	0.28091	0.00899	0.14678	0.08210	0.04304
9	0.25088	0.00497	0.14453	0.07874	0.02264
10	0.26841	0.00351	0.15362	0.08228	0.02900
	0.28447	**0.00673**	**0.14772**	**0.09089**	**0.03584**

Query 8

Run	Response	Pre	Evaluation	Post	Rest
1	0.26936	0.00668	0.11486	0.11463	0.03319
2	0.25325	0.00557	0.11571	0.08230	0.04968
3	0.23601	0.00346	0.11410	0.08253	0.03591
4	0.22069	0.00347	0.11344	0.07460	0.02917
5	0.25119	0.00446	0.11737	0.09225	0.03712
6	0.27127	0.00555	0.11421	0.09951	0.05198
7	0.24169	0.00931	0.11317	0.07505	0.04416
8	0.27539	0.00692	0.11475	0.10039	0.05333
9	0.26417	0.00655	0.11473	0.08923	0.05366
10	0.24722	0.00746	0.11254	0.09184	0.03539
	0.25427	**0.00583**	**0.11437**	**0.08914**	**0.04259**

Query 9

Run	Response	Pre	Evaluation	Post	Rest
1	0.25499	0.00654	0.10629	0.10185	0.04032
2	0.21863	0.00565	0.09676	0.08598	0.03024
3	0.26027	0.00625	0.09909	0.09936	0.05557
4	0.24285	0.00632	0.09855	0.09321	0.04478
5	0.21004	0.00575	0.09699	0.07350	0.03381
6	0.22793	0.00613	0.09687	0.08770	0.03724
7	0.20496	0.00815	0.09695	0.07234	0.02752
8	0.18958	0.00331	0.09871	0.07495	0.01260
9	0.26230	0.00705	0.09929	0.09947	0.05649
10	0.25108	0.00710	0.09757	0.10604	0.04037
	0.23385	**0.00635**	**0.09800**	**0.08950**	**0.03873**

PostgreSQL					
	Query 10				
Run	**Response**	**Pre**	**Evaluation**	**Post**	**Rest**
1	17.45132	0.00592	17.30427	0.09023	0.05090
2	17.16897	0.00811	17.03563	0.09559	0.02965
3	16.96083	0.00617	16.81868	0.09157	0.04442
4	17.51030	0.00551	17.38810	0.08371	0.03299
5	18.19105	0.00404	18.08939	0.07500	0.02262
6	20.45773	0.00740	20.33683	0.07553	0.03797
7	17.98668	0.00595	17.88063	0.06489	0.03521
8	17.08209	0.00453	16.94645	0.08443	0.04668
9	17.99562	0.00472	17.88863	0.06432	0.03796
10	17.76087	0.00556	17.61375	0.09141	0.05015
	17.64336	**0.00572**	**17.51836**	**0.08210**	**0.03938**

Neo4J

Query 1

Run	Response	Pre	Evaluation	Post	Rest
1	6.22870	0.00532	5.64472	0.53210	0.04655
2	6.20025	0.00980	5.64987	0.50950	0.03108
3	6.29305	0.00751	5.72733	0.52436	0.03385
4	6.30937	0.00655	5.76002	0.51421	0.02859
5	6.44997	0.00679	5.61814	0.78894	0.03609
6	6.13331	0.00517	5.61193	0.48319	0.03302
7	6.10146	0.00537	5.52215	0.52012	0.05382
8	6.10503	0.00658	5.56254	0.50075	0.03516
9	6.09878	0.01098	5.53299	0.53867	0.01613
10	6.26495	0.00815	5.71940	0.49857	0.03882
	6.20452	**0.00701**	**5.63337**	**0.51729**	**0.03540**

Query 2

Run	Response	Pre	Evaluation	Post	Rest
1	4.76962	0.00792	4.56656	0.14271	0.05242
2	4.55876	0.00781	4.40641	0.10918	0.03536
3	5.32421	0.00689	5.20476	0.08946	0.02310
4	5.23145	0.00732	5.09545	0.09648	0.03219
5	4.82951	0.00459	4.70263	0.09366	0.02862
6	5.10365	0.00554	4.96893	0.09713	0.03205
7	5.29354	0.00702	5.16852	0.08430	0.03369
8	4.99728	0.00812	4.85528	0.09719	0.03669
9	5.48760	0.00939	5.31468	0.12355	0.03997
10	4.44350	0.00421	4.31318	0.09873	0.02738
	5.01350	**0.00690**	**4.87107**	**0.10067**	**0.03325**

Query 3

Run	Response	Pre	Evaluation	Post	Rest
1	0.13021	0.00621	0.00774	0.08742	0.02884
2	0.15086	0.00949	0.00832	0.09924	0.03381
3	0.13996	0.00645	0.00370	0.08128	0.04852
4	0.12159	0.00435	0.00703	0.07952	0.03069
5	0.12519	0.00605	0.00335	0.08117	0.03461
6	0.13306	0.00339	0.00923	0.09781	0.02264
7	0.15868	0.00884	0.00709	0.10022	0.04253
8	0.21518	0.00816	0.00698	0.16568	0.03436
9	0.14616	0.00572	0.00899	0.09568	0.03577
10	0.14813	0.00727	0.00815	0.08664	0.04607
	0.14153	**0.00663**	**0.00725**	**0.09118**	**0.03584**

Neo4J

Query 4

Run	Response	Pre	Evaluation	Post	Rest
1	0.18669	0.00642	0.01864	0.11004	0.05159
2	0.12536	0.00432	0.00422	0.08209	0.03473
3	0.13326	0.00468	0.00381	0.09347	0.03129
4	0.12835	0.00867	0.00812	0.07407	0.03749
5	0.12815	0.00654	0.00365	0.08512	0.03283
6	0.16921	0.01018	0.01016	0.09857	0.05029
7	0.13979	0.00506	0.00771	0.09384	0.03317
8	0.13416	0.00998	0.00376	0.07496	0.04546
9	0.14765	0.00365	0.01051	0.10851	0.02498
10	0.14419	0.00774	0.00917	0.09647	0.03081
	0.14060	**0.00668**	**0.00718**	**0.09163**	**0.03701**

Query 5

Run	Response	Pre	Evaluation	Post	Rest
1	1.84835	0.00465	1.70140	0.11043	0.03187
2	1.63533	0.00870	1.46772	0.12331	0.03561
3	1.61646	0.00398	1.49214	0.09463	0.02571
4	1.66003	0.00705	1.46912	0.16273	0.02112
5	1.70522	0.00992	1.48901	0.15675	0.04953
6	1.68035	0.00791	1.47846	0.14284	0.05113
7	1.66475	0.00509	1.51152	0.10595	0.04219
8	1.66185	0.01125	1.46062	0.13281	0.05716
9	1.61354	0.00651	1.44778	0.10671	0.05254
10	1.59861	0.00610	1.43692	0.11889	0.03670
	1.65469	**0.00699**	**1.47705**	**0.12471**	**0.04066**

Query 6

Run	Response	Pre	Evaluation	Post	Rest
1	0.21809	0.01038	0.06270	0.10851	0.03651
2	0.21818	0.00712	0.06514	0.10883	0.03709
3	0.18700	0.00400	0.06100	0.10000	0.02200
4	0.25800	0.00700	0.08100	0.11600	0.05400
5	0.22884	0.00607	0.06775	0.11291	0.04210
6	0.23879	0.00589	0.06672	0.13861	0.02757
7	0.21956	0.00905	0.08303	0.09794	0.02953
8	0.21966	0.00409	0.06267	0.11511	0.03779
9	0.24787	0.00704	0.05954	0.15032	0.03097
10	0.18669	0.00874	0.06089	0.08970	0.02736
	0.22225	**0.00688**	**0.06598**	**0.11224**	**0.03361**

Neo4J

Query 7

Run	Response	Pre	Evaluation	Post	Rest
1	2.08373	0.00804	1.79260	0.24874	0.03435
2	2.04086	0.01097	1.72558	0.27258	0.03173
3	1.94767	0.00482	1.70316	0.22295	0.01674
4	2.01779	0.00864	1.76185	0.22472	0.02258
5	2.00567	0.00946	1.75821	0.21295	0.02504
6	2.11482	0.00424	1.84420	0.23385	0.03253
7	2.08967	0.00832	1.79190	0.23562	0.05383
8	2.00907	0.00845	1.73943	0.22275	0.03843
9	1.97505	0.01125	1.71094	0.21810	0.03476
10	1.92252	0.00858	1.62664	0.23552	0.05179
	2.02119	**0.00841**	**1.74796**	**0.23028**	**0.03390**

Query 8

Run	Response	Pre	Evaluation	Post	Rest
1	1.39875	0.00696	1.17326	0.17828	0.04025
2	1.39554	0.00422	1.19883	0.16239	0.03010
3	1.38515	0.00638	1.19134	0.16261	0.02483
4	1.44792	0.00910	1.21556	0.17956	0.04369
5	1.41229	0.00788	1.18788	0.17577	0.04076
6	1.40611	0.00468	1.19841	0.17942	0.02360
7	1.36094	0.00426	1.17007	0.16463	0.02198
8	1.50046	0.00804	1.27273	0.17873	0.04096
9	1.40052	0.00867	1.20946	0.15716	0.02522
10	1.43879	0.00423	1.22487	0.19706	0.01263
	1.41063	**0.00639**	**1.19995**	**0.17267**	**0.03096**

Query 9

Run	Response	Pre	Evaluation	Post	Rest
1	1.16608	0.00653	0.96595	0.16046	0.03315
2	1.19694	0.00643	0.99411	0.15767	0.03873
3	1.16485	0.00484	0.98756	0.15928	0.01316
4	1.18176	0.00526	0.98156	0.16476	0.03017
5	1.19065	0.01008	0.99701	0.15582	0.02773
6	1.17334	0.01008	0.99129	0.14078	0.03119
7	1.16143	0.00673	0.97167	0.14876	0.03427
8	1.24251	0.00491	1.02139	0.17007	0.04613
9	1.18991	0.00643	0.98885	0.15323	0.04142
10	1.21644	0.00677	1.01151	0.16877	0.02939
	1.18500	**0.00664**	**0.99044**	**0.15859**	**0.03326**

Neo4J					
	Query 10				
Run	**Response**	**Pre**	**Evaluation**	**Post**	**Rest**
1	0.11712	0.00392	0.03003	0.07200	0.01117
2	0.14166	0.00936	0.01928	0.08402	0.02899
3	0.14894	0.00400	0.02105	0.08627	0.03762
4	0.14114	0.00778	0.01789	0.07958	0.03588
5	0.15504	0.00558	0.01444	0.10115	0.03386
6	0.19590	0.00981	0.01860	0.11902	0.04846
7	0.15564	0.00871	0.01953	0.09370	0.03369
8	0.13176	0.00579	0.01285	0.09428	0.01884
9	0.12381	0.00396	0.01234	0.09535	0.01217
10	0.12268	0.00498	0.01673	0.08420	0.01676
	0.14008	**0.00627**	**0.01755**	**0.08982**	**0.02723**

F. PROCESSING ANALYSIS RESULT SET

	PostgreSQL								
	10			100			1000		
	Total	Transport	Processing	Total	Transport	Processing	Total	Transport	Processing
1	0.07337	0.01328	0.06009	0.09739	0.02963	0.06776	0.09839	0.03731	0.06107
2	0.09045	0.02579	0.06466	0.09739	0.02963	0.06776	0.09505	0.02500	0.07005
3	0.09064	0.02221	0.06843	0.07231	0.01387	0.05844	0.08857	0.02854	0.06003
4	0.08830	0.02765	0.06065	0.07623	0.01415	0.06208	0.08667	0.02162	0.06505
5	0.07006	0.01195	0.05811	0.08068	0.02141	0.05927	0.09090	0.03200	0.05889
6	0.07924	0.02258	0.05666	0.08584	0.02675	0.05909	0.11499	0.03048	0.08451
7	0.09017	0.02461	0.06556	0.09668	0.02350	0.07318	0.09339	0.03190	0.06149
8	0.11028	0.02211	0.08817	0.07695	0.01347	0.06348	0.08525	0.02549	0.05976
9	0.08360	0.02201	0.06159	0.09771	0.02484	0.07287	0.09266	0.03217	0.06049
10	0.09764	0.02175	0.07589	0.10389	0.04638	0.05751	0.09729	0.02172	0.07556
AVG	**0.08668**	**0.02179**	**0.06437**	**0.08861**	**0.02297**	**0.06384**	**0.09286**	**0.02841**	**0.06419**

	Neo4J								
	10			100			1000		
	Total	Transport	Processing	Total	Transport	Processing	Total	Transport	Processing
1	0.510245	0.05065	0.459597	0.514206	0.07910	0.435107	0.612584	0.12501	0.487579
2	0.482300	0.04869	0.433608	0.500257	0.07477	0.425484	0.571104	0.14037	0.430732
3	0.514964	0.05233	0.462633	0.517307	0.06245	0.454854	0.541836	0.11141	0.430429
4	0.501505	0.05159	0.449911	0.494854	0.07404	0.420818	0.527835	0.11437	0.413467
5	0.503664	0.05186	0.451807	0.505878	0.08207	0.423808	0.506604	0.11756	0.389043
6	0.491737	0.04865	0.443088	0.508790	0.07935	0.429437	0.534342	0.11721	0.417135
7	0.509845	0.05863	0.451215	0.488204	0.07333	0.414872	0.509264	0.10952	0.399748
8	0.581887	0.04929	0.532601	0.534891	0.09693	0.437964	0.510833	0.12104	0.389791
9	0.497222	0.05034	0.446877	0.497328	0.07327	0.424059	0.538981	0.11277	0.426208
10	0.492819	0.04840	0.444418	0.533490	0.07194	0.461553	0.503429	0.11854	0.384891
AVG	**0.502750**	**0.050425**	**0.451193**	**0.509014**	**0.075984**	**0.431441**	**0.530100**	**0.117238**	**0.412069**

	PostgreSQL								
	10000			100000			1000000		
	Total	Transport	Processing	Total	Transport	Processing	Total	Transport	Processing
1	0.190327	0.12571	0.064619	1.24883	1.10136	0.14747	14.80917	14.03603	0.77315
2	0.174112	0.10964	0.064472	1.20110	1.07075	0.13035	14.81569	14.04282	0.77288
3	0.186707	0.12031	0.066400	1.19891	1.06290	0.13601	14.72911	13.91083	0.81828
4	0.178054	0.11102	0.067031	1.23240	1.09655	0.13585	16.20633	15.45660	0.74973
5	0.186015	0.11881	0.067204	1.24032	1.08363	0.15669	14.97345	14.17002	0.80344
6	0.187445	0.12383	0.063618	1.20626	1.07355	0.13271	14.76686	13.98018	0.78668
7	0.194036	0.12818	0.065855	1.20231	1.06560	0.13671	14.70908	13.95835	0.75072
8	0.219680	0.12812	0.091559	1.20150	1.06933	0.13217	14.68367	13.95190	0.73177
9	0.188713	0.11897	0.069742	1.23439	1.09221	0.14217	14.57182	13.80538	0.76644
10	0.190516	0.11805	0.072467	1.19234	1.06814	0.12419	14.78225	14.00662	0.77562
AVG	**0.187727**	**0.120602**	**0.067224**	**1.21465**	**1.07747**	**0.13668**	**14.78366**	**14.00709**	**0.77233**

	Neo4J								
	10000			100000			1000000		
	Total	Transport	Processing	Total	Transport	Processing	Total	Transport	Processing
1	1.208666	0.21652	0.992146	7.911882	1.17400	6.737877	141.4869978	14.12654	127.3604591
2	1.197533	0.19757	0.999963	8.326111	1.16073	7.165383	145.6299000	17.09198	128.5379181
3	1.202968	0.20816	0.994810	8.289004	1.20106	7.087946	142.2470973	16.14020	126.1068931
4	1.298661	0.22249	1.076174	8.131615	1.17656	6.955058	140.2964461	14.56172	125.7347271
5	1.258582	0.21559	1.042991	8.379109	1.18990	7.189213	140.4074080	14.32599	126.0814149
6	1.372147	0.22978	1.142365	8.617907	1.19033	7.427580	140.3249726	14.32485	126.0001228
7	1.228695	0.21141	1.017281	7.851578	1.19563	6.655944	142.1037471	14.91777	127.1859729
8	1.237235	0.22099	1.016246	8.221832	1.18218	7.039655	143.6705220	17.35308	126.3174431
9	1.248965	0.20285	1.046117	8.020385	1.17813	6.842250	143.3043470	17.34290	125.9614439
10	1.269807	0.22274	1.047068	8.001816	1.16524	6.836580	144.3257718	16.44178	127.8839960
AVG	**1.244197**	**0.215093**	**1.030081**	**8.160219**	**1.181496**	**6.981745**	**142.233858**	**15.643400**	**126.612218**

G. FAULT TOLERANCE RESULT SET

Missing Connection		
	PostgreSQL	Neo4J
T1	0.004139	0.060743
T2		
T3	0.00	0.010286
T4	0.00	10.52725
T5	0.00	0.008336
T6		
T7	0.084530	0.00
T8	0.045114	0.00
T9		
T10	0.195722	0.00

Missing Connection		
	PostgreSQL	Neo4J
T1	0.001429	0.024027
T2		
T3	0.111817	0.00
T4	22.705601	0.00
T5	0.057913	0.00
T6		
T7	0.00	0.022038
T8	0.00	0.010094
T9		
T10	0.00	2.760941

Connection Loss		
	PostgreSQL	Neo4J
T1	0.000947	0.020123
T2	0.080691	0.005577
T3		
T4	0.000000	1.863033
T5		
T6	0.041675	0.000000
T7		
T8	0.156403	0.000000

Connection Loss		
	PostgreSQL	Neo4J
T1	0.001355	0.025314
T2	0.045180	0.022883
T3		
T4	21.860407	0.000000
T5		
T6	0.000000	0.082107
T7		
T8	0.000000	0.028500

Transaction Failure		
	PostgreSQL	Neo4J
T1	0.001079	0.019157
T2	0.001396	0.005303
T3	0.275398	0.000000
T4	0.000000	0.017561
T5	0.068518	0.004775