Bridge-DB

Query Optimization in a Multi-Database System

Rune Ettrup
Department of Computer Science
Aalborg University
rettru09@student.aau.dk

Lisbeth Nielsen
Department of Computer Science
Aalborg University
lniel10@student.aau.dk

ABSTRACT

In this paper we present a distributed database system, called Bridge-DB. This system focuses on using multiple data sources without any prior knowledge of the underlying database architecture as well as simplify interaction with multiple database systems.

Bridge-DB has its own query language BQL which supports all CRUD operations. It is connected to PostgreSQL and Neo4J and through the modular design of the system it supports any type of storage mechanism through the implementation of a database driver module.

Our main contribution is the implementation of a cost-based optimizer using a combination of a dynamic and black box cost model to determine which database a query should be executed on, or whether the query should be enumerated, and executed on multiple databases after which the optimizer does post-processing of the results to fulfill the query.

The new solution has been tested against two different datasets each with a bias towards Neo4J and PostgreSQL respectively and a combination of both in order to test the effectiveness of the cost model. Based on response times, data traffic and overhead of Bridge-DB we show that through the use of our cost model we gain higher performance on response times at the cost of an increase in data traffic between Neo4J and Bridge-DB.

1. INTRODUCTION

With the Not only SQL (NoSQL) movement came many new database models, each having a speciality that the dominant database model, the relational model, cannot handle as efficiently [14]. For example, many NoSQL database models are more flexible since they have no rigid schema and it is easier to add more servers. A few concrete examples are the key-value databases demonstrate good performance when handling temporary unstructured data, graph databases handle relationships as first-class citizens and column databases are better for storing historical data for business analysis. Even though relational databases have gained more competition, they are still dominant database.

Today, as a result of this development more and more companies use multiple database systems to satisfy their own and their customers' needs. In these systems each database is used, such that the advantages are exploited and the weaknesses are covered by another database system. This is also referred to as polyglot persistence [14, 18].

A disadvantage of using multiple databases is the increase of complexity in the model layer [18]. First, there are no single database access point for multiple heterogeneous database because typically Object/Relational Mapping (ORM) libraries only connect to a single database. Secondly, each Database Management System (DBMS) has its own data declaration and manipulation language, so developers need to learn multiple query languages. In addition, the developers also need to know, how the data has been partitioned into the different databases, such that when they query a database, it contains the wanted data. Finally, when querying the different databases we also risk getting the result in different format which also increases the complexity further.

A solution to this problem is to make a multi-database system which incorporates a simplified method to interact with multiple heterogeneous database management systems and then presents a unified data declaration and manipulation language to the user. Such a system is presented in our previous work [8] where we presented a system called Bridge-DB.

Bridge-DB

Bridge-DB focuses on creating a middle-ware layer between heterogeneous databases with the purpose of leveraging the advantages of these databases. Bridge-DB uses PostgreSQL [10] and Neo4J [21], as they represent two different database models, query languages, and they are also clear opposites in most use cases. Bridge-DB has been designed as a modular system allowing a user to add or remove modules, e.g. optimizer module or database connector module [8].

During the evaluation of Bridge-DB it was demonstrated that it is a viable solution to use a middle-ware layer to connect heterogeneous databases, since the overhead is low compared to querying the databases directly, in most cases less than 50 ms [8].

Bridge-DB has its own query language called Bridge-DB Query Language (BQL) which is inspired by SQL, but with extensions to allow graph operations. However, BQL is less expressive than SQL and does not support nested queries or parenthesis used to control the conditions evaluation. Bridge-DB is meant to be a distributed database system, but BQL does not support any write operations to the databases, which defeats the purpose. Therefore in the extended version of Bridge-DB all Create, Read, Update and Delete (CRUD) operations have been implemented, and the language constructs for read queries have been extended to make more complex queries that can be a challenge for the optimizer.

Bridge-DB implements a heuristic optimizer to determine the database which a query should be executed on. This is an insufficient solution if Bridge-DB connects to several databases, due to the heuristic optimizer could potentially find multiple databases indistinguishable from each other, or if Bridge-DB extends its functionality with support for more complex queries. A cost model would be a better solution to do query optimization based on it does not focus on which databases uses a specific paradigm or which database is compatible with certain parts of a query, but is more focused on response times which are the reason we have replaced the heuristic model with a cost model. To support the cost model, Bridge-DB implements a plan enumerator capable of decomposing the individual queries, in the cases the cost model deem it necessary to decompose a query among multiple databases to achieve better performance.

The query translation from BQL to especially Cypher is not fully supported. The traverse and reachable queries could be translated, given the values of two internal Cypher indexes of two nodes and then make the Cypher queries from this. Instead we want to find the nodes via a property, such that we can make a property that also uniquely identifies the node in Bridge-DB. The translation of SQL-like queries to Cypher also has a limited implementation where the translation only succeeded with simple selection queries that did not contain a join operation. In the extended version of Bridge-DB this has been improved.

Bridge-DB did not store information about the data in the databases, but only a simple mapping solution from SQL indexes to Neo4J indexes. This mean that Bridge-DB could not validate a query or translate a query which could be executed on the external databases. It is also difficult to make a cost model without any information about the schema and since Bridge-DB works with PostgreSQL then a schema declaration method is necessary. Due to the advantages of having a schema within the system, the schema is stored in Bridge-DB.

The REST API of Bridge-DB could be optimized with an TCP Socket implementation. By doing this, we can limit the overhead of communication between the client and Bridge-DB as well as allowing bi-directional communication, without the need to reconnect with each message transmission, as required with the REST API. Bi-directional communication allows Bridge-DB to supply the client with intermediate information such as optimizer decisions and query execution progress. This solution also allows us to process incoming query results on the client before the entire result is received, limiting the blocking-time a query can cause.

Novel Contributions

In summary our novel contributions in this paper are to extend the original design of Bridge-DB with some additional functionality and modules to improve the issues of the original design.

First, the REST API has been replaced by a TCP Socket based module that contains a protocol to wrap communication between Bridge-DB and the client.

A new functionality in Bridge-DB is the schema declaration, such that a schema can be forced on the external databases. Also the schema can advantageously be used during query optimization and translation.

The third contribution is to extend BQL such that read operations have become more expressive and all of the CRUD operations are implemented. This also leads to the reimplementation of the SQL and Cypher translators, so they are able to translate the extended version of BQL into SQL and Cypher, respectively.

The main contribution is the implementation of an optimizer module with a cost model capable of leveraging the strengths and weaknesses dynamically of the individual connected databases, based on continuous measurements of query execution times. This allows Bridge-DB to execute queries using an estimated optimal decomposition of the original query on multiple databases and post-process the partial results from the individual databases on Bridge-DB to fulfill the query. By decomposing we can execute individual parts on different databases to take advantage of the strengths of the connected databases.

With these contributions Bridge-DB can be considered an operational distributed database system which reduces the complexity of using the graph database Neo4J and the relational database PostgreSQL in unison. For the remainder of this paper Bridge-DB and BQL will refer to the extended versions.

Structure of the paper

The remainder of this paper is structured as follows: in Section 2 we will look at some data and cost models that are the foundation of Bridge-DB. Bridge-DB's architecture, schema declaration, and BQL are explained in Section 3. The translation process from a BQL query to a Cypher query is presented in Section 4. Section 5 explains the optimizer and cost-model of Bridge-DB in detail. We evaluate the implementation of Bridge-DB and its optimizer in Section 7. Finally, in Section 8 we conclude on the paper and explain possible improvements of the system.

2. BACKGROUND AND PRELIMINARIES

As in the previous version of Bridge-DB, Neo4J and PostgreSQL are used as they present two different types of databases, and are designed to solve different types of problems. Both databases are used in the same manner in the current implementation of Bridge-DB, but to improve on them, we need a better understanding of the data models.

To improve on the heuristic optimizer from the previous version of Bridge-DB we have chosen to implemented a cost based optimizer and therefore we need to understand the different approaches to making a cost-model for a heterogeneous multi-database system.

2.1 Data models

Both a relational and graph database are used in Bridge-DB, so to get a common understanding of the two models some concepts are presented in this section. We also show how a relational designed schema can be represented in a graph such that we can make a schema on Bridge-DB that can be enforced on both system.

Relational model

In [7] Edgar F. Codd suggested the relational model which has been the foundation for relational databases. In Figure 1 is the relational model represented as a table, and it also shows some of the terminology mentioned below.

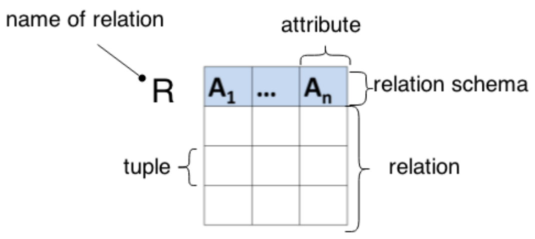


Figure 1: A representation of the relational model

Relational database terminology and concepts will be used throughout the paper, therefore it will be clarified in this section. A relation is typical also referred to as a table, and it contains all the data. The relation has a name which is called relation name or table name. A row is called a tuple, and it contains the data of a single row in the relation. A column is also called an attribute and the head of the column is the attribute name. [20]

A relational database consists of a collection of relations. The database schema is the blueprint of the database, so each relation schema is also a part of the database schema. The relation schema contains information about the constraints of an attribute such as its data type, uniqueness of the attribute, etc. It also contains information about keys such as the primary key which identifies each tuple in a relation, or the foreign key which references an attribute in a relation.

In a relational database the foreign keys are used to make relationships between relations. These relationships have three types: one-to-one, one-to-many, or many-to-many. In Figure 2 is a conceptual representation of a database schema, and how this schema would be represented in a relational or graph database. There is a one-to-many relationship between *person* and *user* where the foreign key is in the *user*. A similar solution is made for a one-to-one relationship. The relation between two *person* is a many-to-many and a new relation is need to contain the relationship.

The relational databases has a strict schema, and the DBMS

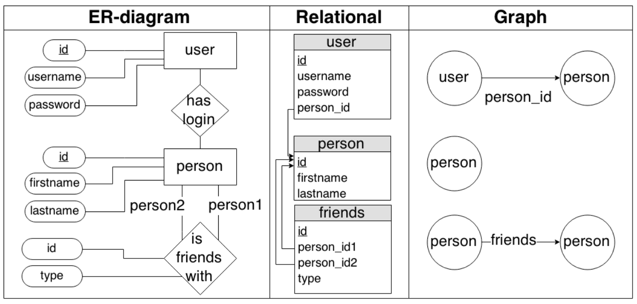


Figure 2: A conceptual representation of a schema and its representation in a relational and graph database

handles constraints and foreign-key dependencies. This also means that the DBMS handles constraints introduced by using foreign-key references. For example a foreign-key must reference an existing value in the referenced table, but if the tuple with this value should be deleted then the DBMS can take some predefined actions as set the reference to null or another value, or do not allow the deletion of the tuple. This also means that when executing an update or delete query through Bridge-DB, we do not have to take the foreign-key dependency into account when writing the SQL query.

Graph model

During the last decade graph databases have made their entry on the commercial market like other data models in the NoSQL movement. Graph database are based on graph theory, and is optimized to perform graph operations like finding a path between two nodes. These databases can use different graph models, but this paper will only focus on Neo4J's data model, which is the labelled property graph model, see Figure 3.

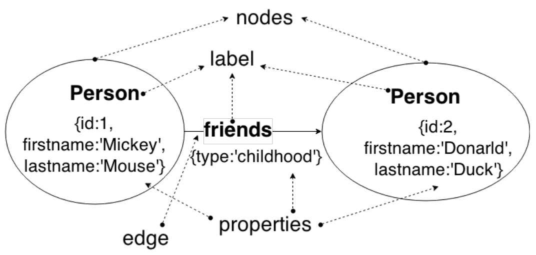

nodes
label
Person
{id:1, firstname:'Mickey', lastname:'Mouse'}
friends
{type:'childhood'}
Person
{id:2, firstname:'Donarld', lastname:'Duck'}
edge
properties

Figure 3: A representation of the labelled property graph model

There are two types of elements in a graph. One is a node or vertex, and the other is an edge which makes a relationship between nodes. The edges are directed in Neo4J which means an edge always has a stated begin and end node, but it can be queried as if it was an undirected graph. Nodes and edges can also have attributes or properties, as it is called in a graph. A property is a key-value representation of the property name and its value. [4]

Both a node and an edge can have several labels. A label represents a role in the domain of databases and it is used to group nodes or edges together, which typically have some properties in common. Labels can also be used to make constrains and indexes for a property. [4]

Figure 2 shows how the *person*, *user*, and *friends* from the ER-diagram are represented in a graph. In the figure only the labels of the nodes and edges have been included to simplify the graph. But the nodes and the edge *friends* have the same properties as in the ER-Diagram while the edge labelled *person_id* does not have any properties.

Differently from the relational database, Neo4J has an optional and flexible schema which is based on the use of labels. For Bridge-DB this leads to more control when translating queries. Because ideally the foreign-key or edge dependency should be taken into account when translating an update or delete query to Cypher such that it will do the same as the relational DBMS.

From a relational to a graph model

In Bridge-DB we introduce a schema declaration method which resembles the relational schema, but it should also be converted into a graph schema. On Neo4J's webpage [2] is some guidelines of how a relational database can be converted into a graph database in Neo4J. By using some of the same principle, we can convert a relational database schema into a graph schema.

All tables in the schema are categorized as either a join table or a data table which is defined in Definition 1 and 2. This results in a one-to-one mapping between these table types, and the graph schema's nodes and edges.

DEFINITION 1. *A join table is a relation that is not referenced by another table, it contains exactly two foreign-keys and it can also have other attributes.*

DEFINITION 2. *A data table is a relation that contain any other case than the join table. This means that the relation can have none or several foreign-keys and it can be referenced by another table.*

Each data table is made into a node where table name becomes the label of the node and the attribute names are converted into property keys. Each property key is connected to an object containing information about the attribute such as its data-type, constraints, and indexes which have been declared in the original schema, as explained in Section 3.2. Each foreign-key in a data table becomes an edge with the foreign-key name as a label. Each join table is made into an edge between two nodes that are referenced by the two foreign-keys. The attributes of the join table will be made into properties of the edge and the table name become the label name of the relationship.

By using this method a join table covers a many-to-many relationship, while the data-table covers the entities, one-to-one and one-to-many relationships explained previously as well as other n-ary relationship types.

This graph schema is used when translating queries from BQL to Cypher such that we can handle foreign-key references, as will be explained in Section 4.

2.2 Cost models

When querying a Multi Database Management System (MDBMS) the query could be executed on some or all of the database source, depending on the optimizer. To optimize the execution of a query a cost model is needed. MDBMS cost models are often divided into three methods called Blackbox [9, 26], Customized [17, 27], and Dynamic method [13, 30].

Black Box Method

The black box model is an approach that considers each DBMS as a black box and by running some test queries on each database, the information for the cost model can be collected. [15]

In the CORDS project [28] a cost model is made by probing the individual DBMSs in order to determine cost information. An extension of this method is proposed in [29] which focuses on having sample queries. At the beginning, all queries are classified into homogeneous classes according to a set of criteria. Then some queries from each class are executed and measured to derive cost information.

The black box methods are inhibited by the probe and sampling queries that can take up resources when executed. To avoid this, [5] implements a progressive learning cost model, similar to some of the ideas in the dynamic method. Overall, the black box method has a general problem being that it is not necessarily able to catch the individual specifics of the individual database.

Customized Method

The customized method focuses on the idea that all DBMSs are too different to be represented by a unique cost model, which the black box method suggests. This method is focused on implementing a specialized cost model for the individual databases and the aggregated result of these models creates an overall cost model applicable on the system.

This is typically implemented in a mediator-wrapper architecture where the wrapper contains the specific cost information of its database and it can implement its own cost model. The wrappers have a common interface which the mediator can use and the mediator need to integrate cost information into the query optimizer. [15]

[17] presents a framework of making a customized cost model where they demonstrated the simplicity and effectiveness of this method as a simple implementation resulted in significant improvements.

Dynamic Method

The dynamic approach focuses on fixing a problem with both the customized and black box model which is they both assume a stable database environment. Instead the dynamic approach makes a cost model from monitoring the run-time behaviors of each database [15]. [16] describes the three factors in determining cost for each database and each of them focusing on how dynamic they are. The first factor consists of CPU load, I/O throughput, and available memory. The second factor consists of configuration parameters, physical data organization and database schema. The third factor consists of DBMS type, database location, and CPU speed. Each going from a fast changing factor to a stable factor respectively. So by observing these factors, a cost model can be made and it would be more accurate than the black box and customized method. However, the dynamic approach will also result in some overhead when querying a database [15, 16].

In [25] a method is proposed which takes some ideas from the black box method and implements them to account for the more dynamic nature. The user queries are used as sample queries which avoids the issue of sample queries causing additional overhead when executed.

A fusion of all three methods is described in [24] where the author attempts to predict execution time of a query on both concurrent and dynamic database workloads. This is done by using knowledge of the built-in optimizers cost model. The query is decomposed into its base constructs and the execution time of them individually is compared to previous queries which is similar to the dynamic method as queries are continuously evaluated as they are executed.

In Bridge-DB we want to use some concepts from the black box and dynamic approach. Mainly, we will implement a dynamic optimizer based on the query decomposition ideas from [24], but we will initialize the cost model by doing some probing of each database with some measurement queries as in the black box approach. We choose this cost model because the optimizer would then be able decompose a query and execute parts of it on different databases and thereby exploit the advantages of each database.

3. BRIDGE-DB DESIGN

Bridge-DB has been re-implemented and several changes have been made, so in this section we present some of the design decisions of Bridge-DB.

3.1 Architecture

The architecture of Bridge-DB is shown in Figure 4. It is a client-server architecture where a client can write a query in BQL via the *Querybuilder* and send it, wrapped in JSON to the server. The communication protocol is a simple TCP-socket implementation and therefore will not be discussed in detail, but it can be found in Appendix A.

The server has modules for translating the JSON wrapped query into a *query object* which the optimizer can manipulate. Eventually, the optimizer passes the object to a connector which translates it into the query language of the target database, and finally executes the query on the external database.

The optimizer contains several modules such as a cost model and a plan enumerator, which are explained in detail in Section 5. The optimizer uses an internal database to save information for the cost model, and the declared schema which is forced onto the external databases. If the optimizer decides that a query should be distributed across multiple databases then the optimizer also needs to join the results from the databases before sending the result to the user.

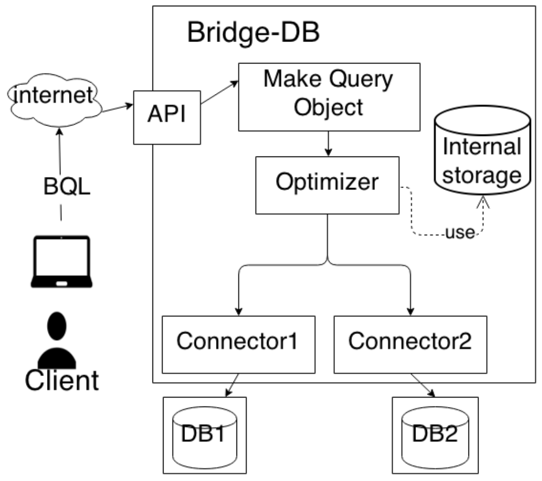


Figure 4: Architecture of Bridge-DB

Connector

For each external database that should be connected to Bridge-DB there must be a connector. The sequence diagram in Figure 5 describes, how a connector works. The optimizer decides which database should execute the query and selects its respective connector, and passes the query object to it. First, the connector needs to have a connection to the target database. Then the query is translated into the target query language, but to do this, the connector needs the schema to validate and make the query. After the query has been translated, it is sent to the database which then executes it and returns the result.

The connector makes the result into a common format for all connectors and then returns it to the optimizer.

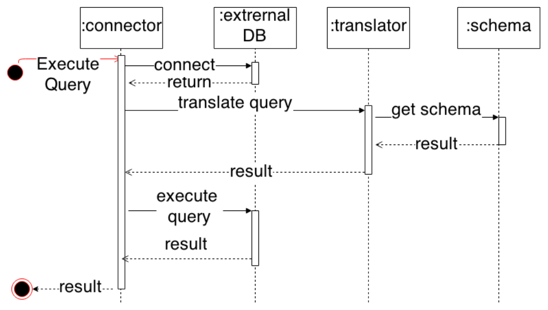


Figure 5: Sequence Diagram of a Connector

Currently in Bridge-DB there are two external databases, Neo4J and PostgreSQL. The implementation of the translator module of PostgreSQL is simple since the schema and BQL are SQL inspired, see Appendix B for more details. The translator module of the Neo4J connector is more complex due to the differences in the schema representation and its query language Cypher. See more of translating into Cypher in Section 4.

3.2 Database schema

We chose to design a way of declaring schemas inside Bridge-DB, to move information of the schema layout from the individual databases and into Bridge-DB.

When a schema is declared in Bridge-DB, it will be imposed onto the individual databases. This is done through the generation of the schema on the database servers, like PostgreSQL, and via the validation of the input query's compatibility with the schema on schema-less database servers, like Neo4J.

By declaring the schema in Bridge-DB, it allows for easier validation of the incoming query. The declaration of the schema can also assist the optimizer in making decisions on the most efficient query execution plan (QEP), all without retrieving information from the individual databases which is described in Section 5.

The language is designed as seen in Listing 1. It is called Bridge-DB Schema Language (BSL) and is also inspired from SQL. All changes to the schema are named a migration and contain an *up* and *down* method. They are similar to version control as the *up* method handles the transition from a previous schema to a new schema and the *down* method handles the transition from a current schema to a previously declared schema.

This is inspired by the PHP framework Laravel [3], but with a rather different storage mechanism and underlying implementation. By declaring the schema in this way, it is possible to have version control of the schema.

```
namespace schemaDecleration\Migrations;

class CreateDatabase extends Migration {
 public function up() {
  \schemaDecleration\Schema::create("person",
  function(\schemaDecleration\Blueprint
      $table) {
   $table->increments("id")->primaryKey();
   $table->string("firstname");
   $table->string("lastname");
  });

  \schemaDecleration\Schema::create("user",
  function(\schemaDecleration\Blueprint
      $table) {
   $table->increments("id")->primaryKey();
   $table->integer("person_id")->unsigned();
   $table->foreign("person_id")->references("
       id")
   ->on("person")->onDelete("cascade");
   $table->string("username")->unique();
   $table->string("password");
  });

  \schemaDecleration\Schema::create("friends"
      ,
  function(\schemaDecleration\Blueprint
      $table) {
   $table->increments("id")->primaryKey();
   $table->string("type");
   $table->integer("person_id1")->unsigned();
   $table->foreign("person_id1")->references(
       "id")
   ->on("person")->onDelete("cascade");
   $table->integer("person_id2")->unsigned();
   $table->foreign("person_id2")->references(
       "id")
   ->on("person")->onDelete("cascade");
  });
 }

 public function down() {
  \schemaDecleration\Schema::drop("friends");
  \schemaDecleration\Schema::drop("user");
  \schemaDecleration\Schema::drop("person");
 }
}
```

Listing 1: Bridge-DB Schema Language (BSL)

Version control becomes vital as Bridge-DB creates the declared schema on compatible servers and enforces the schema internally for schema-less database systems. Therefore changes to the schema can require modifications to the stored data which Bridge-DB needs to handle internally. When a migration is executed through the *up* and *down* methods, it allows the developer to declare, how the changes to the stored data should be done both when upgrading and downgrading.

The declared schema in Listing 1 will act as an example describing the simple schema from Figure 2 with *person*, *user*, and *friends*. As can be seen from the example when declaring a schema, we use many of the same constructs as SQL such as declaring data-types, primary, and foreign keys, etc.

This schema is compatible to a relational database like PostgreSQL, while Bridge-DB must impose the schema on Neo4J internally in most cases. Neo4J can handle constraints as *unique* and indexing as making indexes on primary keys, while constraints on foreign-keys must be handled by Bridge-DB. These foreign-key constraints for Neo4J have not been fully implemented, therefore we limit the constraints only to include the cascade constraint.

The cascade constraint on Neo4J is enforced through checking declared cascade constraints in the declared schema on Bridge-DB. If a there a declared cascade constraint, a new query is generated which enforces this constraint.

3.3 BQL

BQL is a high-level query language and it is highly inspired from SQL, but in its previous version it only included read operations. BQL has therefore been extended in Bridge-DB to support all of the CRUD operations. A BQL query is written via a *QueryBuilder*, and it is made into an object in Bridge-DB that can be translated into Cypher and SQL.

BQL has been expressed in Extended Backus-Naur Form (EBNF), and in Table 1 is a list of symbols and constructs that is used to express the EBNF of BQL which is presented in Listing 2. The non-terminals are primarily grouped by the query type, but if a non-terminal is used by multiple types then it is presented at the first occurrence.

```
<Query> ::= <CreateQuery> | <UpdateQuery>
        |   <DeleteQuery> | <ReadQuery>;

<CreateQuery> ::= <table> (<values>
        | <column> <sub-query>);
<table> ::= "string";
<attribute> ::= "string"
        | <table> , '.', "string"
        | <alias> , '.', "string" ;
<values> ::= <column> <value>;
```

Usage	Symbol
definition	::=
termination	;
alternation	|
option	[...]
concatenation	,

Usage	Symbol
grouping	(...)
nonterminal	<...>
terminal string	' ... '
terminal empty	empty
terminal	string numeric char array

Table 1: Symbols and constructs used in the EBNF of BQL

```
11  <value> ::= <mixed> | <mixed> <value>;
12  <column> ::= <attribute>
13          | <attribute> <column>;
14  <sub-query> ::= '(' <ReadQuery> ')';
15  <mixed> ::= "numeric" | "string" | "char";
16
17
18  <UpdateQuery> ::= <table> <values> [<where>];
19  <where> ::=  <conditionConstructs>
20          | <conditionConstructs> <where>;
21  <conditionConstructs> ::=  <type> <condition>
22          | <type> <parenthesis-sub-query>;
23  <type> ::= 'AND' | 'OR' | "empty";
24  <parenthesis-sub-query> ::=  '(' <where> ')';
25  <condition> ::= <condLeft> <operator>
26          <condRight> | <exists> <sub-query> ;
27  <condLeft> ::= <attribute>;
28  <condRight> ::= <mixed> | <attribute>
29          | array |  <sub-query>;
30  <operator> ::= 'LIKE' | 'NOT LIKE' | 'IN'
31          | 'NOT IN' | '=' | '<' | '>' | '>='
32          | '<=' | '<>' | '!=' | '!<' | '!>';
33  <exists> ::= 'EXISTS' | 'NOT EXISTS';
34
35
36  <DeleteQuery> ::= <table> [<where>];
37
38
39  <ReadQuery> ::= <select> [<from> [<where>]]
40          | <traversal> | <reachable>;
41  <traversal> ::= <sub-query> <sub-query>;
42  <reachable> ::= <sub-query> <sub-query>;
43  <select> ::= <attribute>
44          | <attribute> <select>;
45  <from> ::= <fitem> | <fitem> <from> ;
46  <fitem> ::= <table>  [<alias>]
47          |  <sub-query> [<alias>];
48  <alias> ::= "string";
```

Listing 2: Extended Backus-Naur Form of BQL

As can be seen from the statement at line 1 in Listing 2 a query can be a create, read, update or delete query. The create is presented in the second grouping at line 4 in Listing 2, the update is presented in the third grouping and then there is a line with the construct for a delete query. The read query is presented in the final grouping and it shows that there are three types of read queries: traversal, reachable, and an SQL-like query.

As can be seen from the *conditionConstructs* and *condition* non-terminals, it is possible in the *where* clause which means that a sub-query or parenthesis-sub-query are in the query. As can be seen a *parenthesis-sub-query* is an extra *where* clause inside the parent query which makes it a special sub-query. A normal *sub-query* is an extra query inside the parent query, but the sub-query can only be a *read query.*

QueryBuilder

When a client needs to build a BQL query, they can use the *QueryBuilder* class which is written in PHP. With the *QueryBuilder* class, the query can be written and sent to Bridge-DB. In Listing 3 are a few examples of how to make an insert, update, or delete query using the *QueryBuilder.* In the first example a *person* is inserted with given values for *firstname* and *lastname.* In the second query we update the *firstname* of a *person* is changed and in the last query we delete all *persons* with a given *firstname.*

```
//example of a create/insert query
$queryBuilder->insert('person')->values('
    firstname','Mickey')->values('lastname','
    Mouse')->send();

//example of a update query
$queryBuilder->update('person')->values('
    firstname','Minnie')->where('lastname','=
    ','Mouse')->where('firstname','=','Mini')
    ->send();

//example of a delete query
$queryBuilder->delete('person')->where('
    firstname', '=', 'Mickey')->send();
```

Listing 3: BQL examples of how to make a read, update and delete query

Additional constructs for a read query have been included in BQL. For example sub-queries and parenthesis for controlling the logic in the *where* clause have been implemented into BQL, see the 2. However, a query that contains a sub-query is only fully supported on PostgreSQL. Using parenthesis in a query has been implemented as a special sub-query and it is supported on both PostgreSQL and Neo4J. Listing 4 shows how a read query can be made in BQL when using sub-queries or parenthesis. The first query uses a sub-query to find all *firstname* of *persons* that have a *lastname* in common, and then in the main query find all persons with a *firstname* which also appear in the result of the sub-query. The second query show how a parenthesis sub-query we find all *person* with the *lastname* Mouse and where the *firstname* is either Mickey or Minnie.

A more detailed description of how to make a create, read, update, or delete query using the *QueryBuilder* can be found in Appendix C.

```
//example of a read query with a subquery
$queryBuilder2->select('firstname')->from('
    person)->where('lastname','=','Mouse');
$queryBuilder->select('*')->from('person')->
    whereIn('firstname',$queryBuilder2->
    getReadQueryAsObject())->send();

//example of a read query with parenthesis
$queryBuilder2->where('firstname','=','Mickey
    ')->orWhere('firstname','=','Minnie');
$queryBuilder->select('*')->from('person')->
    where('lastname','=','Mouse')->where(
    $queryBuilder2->getReadQueryAsObject()->
    send();
```

Listing 4: BQL examples of using parenthesis and sub-queries in a read query

4. TRANSLATION TO CYPHER

In the previous version of Bridge-DB only a few queries in BQL could be fully translated into Cypher. This was due to the complexity of translating an SQL-like query into Cypher without knowing whether a foreign-key referred to a node or edge. In Bridge-DB all SQL-like queries that do not contain a sub-query can be translated into Cypher.

First, when translating a query object into Cypher it needs to turn the schema into a graph schema. The second step is to do translation pre-processing of the query object, such that it gets a new representation that can easily be translated into Cypher. The final step is the translating into Cypher.

4.1 From General Schema to Graph Schema

As previously mentioned the schema is important when translating a query because data and join tables are represented differently in Cypher as either nodes and edges. Even though, the declared schema in Bridge-DB can be used for this purpose then it would be easier to have a graph representation of the schema. Therefore the schema is converted into a representation of the graph schema, such that the graph will be as described in Section 2.1.

Between the schema and graph schema is a one-to-one correspondence. A table in the schema will correspond to a single edge or node type in the graph schema. In the schema each table can be uniquely identified by its table name and likewise can a table name uniquely identify whether it is a node or an edge in the graph schema. However, with this representation of the graph schema we only represent some of the edges from the graph database. For example, an edge made from a foreign-key in a data table is not represented as an edge in the graph schema. Instead they are represented in the node as a special type of properties, we call foreign-key property. The foreign-key property is not represented as properties in the graph database, but are made into edges during translation.

The graph schema consists of three classes *Graph*, *Node*, and *Edge* which are presented in the class diagram in Figure 6. The class *Graph* generates a graph schema upon initialization, and then it is a container and access point for the nodes and edges of the graph. *Node* and *Edge* contain all data about its properties and foreign-key properties. *IGraphEntity* is a common interface for a *Node* and *Edge* which makes the data accessing easier when translating a query.

When the graph schema is complete then there is a single instance of the *Graph*, which contains two lists, one with instances of *Edge* and one with instances of *Node*. By choosing this solution it is easy to find and retrieve the node or edge. A common problem with this approach, is that a node with foreign-key properties needs to be handled differently from a node without foreign-key properties. This is a reoccurring problem throughout the translation of queries as will be apparent in Section 4.3.

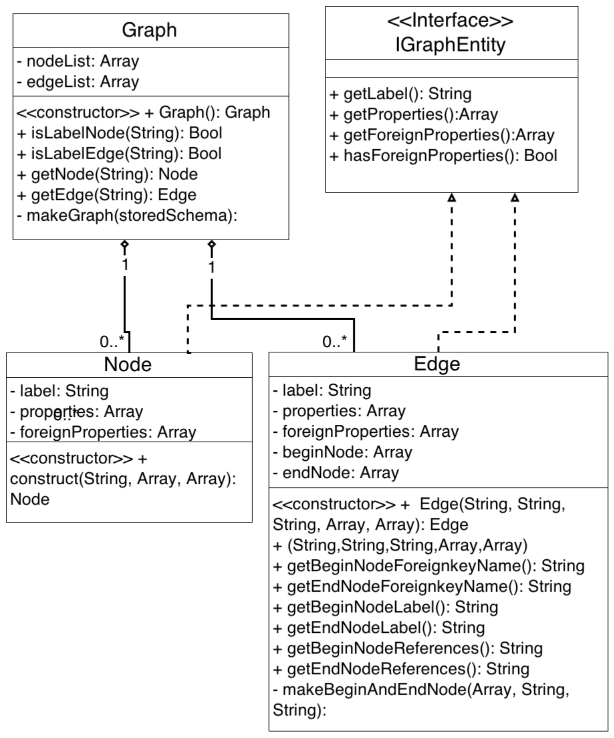

Graph
- nodeList: Array
- edgeList: Array
<<constructor>> + Graph(): Graph
+ isLabelNode(String): Bool
+ isLabelEdge(String): Bool
+ getNode(String): Node
+ getEdge(String): Edge
- makeGraph(storedSchema):
<<Interface>> IGraphEntity
+ getLabel(): String
+ getProperties():Array
+ getForeignProperties():Array
+ hasForeignProperties(): Bool
1
1
0..*
0..*
Node
- label: String
- properties: Array
- foreignProperties: Array
<<constructor>> + construct(String, Array, Array): Node
Edge
- label: String
- properties: Array
- foreignProperties: Array
- beginNode: Array
- endNode: Array
<<constructor>> + Edge(String, String, String, Array, Array): Edge
+ (String,String,String,Array,Array)
+ getBeginNodeForeignkeyName(): String
+ getEndNodeForeignkeyName(): String
+ getBeginNodeLabel(): String
+ getEndNodeLabel(): String
+ getBeginNodeReferences(): String
+ getEndNodeReferences(): String
- makeBeginAndEndNode(Array, String, String):

Figure 6: Overview of the interface IGraphEntity and classes Graph, Edge and Node

4.2 Pre-processing of the Query Object

The update, delete and read query objects can all contain conditions which are used to filter the result by using the *where* construct of BQL. In Cypher the filtering can be done by a WHERE clause similar to SQL and BQL. However, translating conditions of a query object to Cypher is not straightforward, because Cypher uses aliases differently from BQL. Another problem is the constructs parenthesis sub-query, foreign-keys, and patterns, that need to be identified and handled. This puts an extra layer of complexity when translating the query into Cypher and therefore some pre-processing of the *where* clause is necessary.

Pre-processing of Parenthesis Sub-Queries

The first pre-processing action is to make a flat representation of the query such that additional pre-processing of the query object also will be done inside the parenthesis.

The Cypher translator supports the special parenthesis sub-query which means that the conditions of query object can have a hierarchical representation. In Listing 5 is the definition of BQL's *where* construct in EBNF, but it has been adapted to show only the constructs which are supported in the Cypher translator. In the listing it is apparent that a *conditionConstruct* can lead to some recursion which results in a hierarchical representation.

```
<where> ::=  <conditionConstruct>
        | <conditionConstruct> <where>;
```

```
<conditionConstruct> ::=  <type>  <condition>
        | <type> <parenthesis-sub-query>;
<condition> ::= <condLeft> <operator>
        <condRight>;
<parenthesis-sub-query> ::=  '(' <where> ')';
```

Listing 5: Hierarchical Representation

This needs to be converted into a flat representation as shown in Listing 6. This is done by checking for parenthesis sub-queries in the conditions. When a parenthesis sub-query has been found then its conditions are placed between the conditions of the parents query. However, the first and last element of the parenthesis sub-query get an extra parameter with its begin or end bracket.

```
<where> ::=  <conditionConstruct>
        | <conditionConstruct> <where>;
<conditionConstruct> ::= <type> <parenthesis>
     <condition>;
<condition> ::= <condLeft> <operator> <
    condRight>;
<parenthesis>   ::= '(' | ')' | "empty";
```

Listing 6: Flat Representation

The hierarchical and flat representations of the query object's *where* construct have also been illustrated as object diagrams in Figure 7. These object diagrams use the second example from Listing 4 to show the changes in the structure of the object.

Pre-processing of update and delete queries

The pre-processing of update and delete queries is necessary to make BQL aliases compatible with Cypher aliases, and to deal with foreign-key references in the query. The solution presented, is applicable as long as the query only directly refers to one node or edge, as update and delete does.

An alias in BQL, SQL, and Cypher is a temporary name for a table, node or edge. In SQL and BQL aliases are necessary when a table is included more than once in the query. Aliases are necessary in Cypher when a node or edge needs to be referenced more than once.

Aliases are not needed when writing an update or delete query in BQL, but this is not the case for Cypher. Therefore all properties which are referred in Cypher's WHERE clause, need an alias preceding it. In an update or delete query just one table is directly referred to, so by using the table name this alias problem can be solved. An example can be found in Listing 7, where a delete query needs to delete a *user* and its edges.

```
MATCH (user:user)-[r0:person_id]-(person_id:
    person) WHERE person_id.id = 2  WITH
    users OPTIONAL MATCH (user)-[r]-() DELETE
     user,r
```

Listing 7: Examples of alias assignment in a delete query both in BQL and Cypher

The second part of the problem is to handle foreign-keys that are represented in the query and their aliases. A foreign-key is a reference to a specific property in a node. This property and its node will be referred to as the *referred element.*

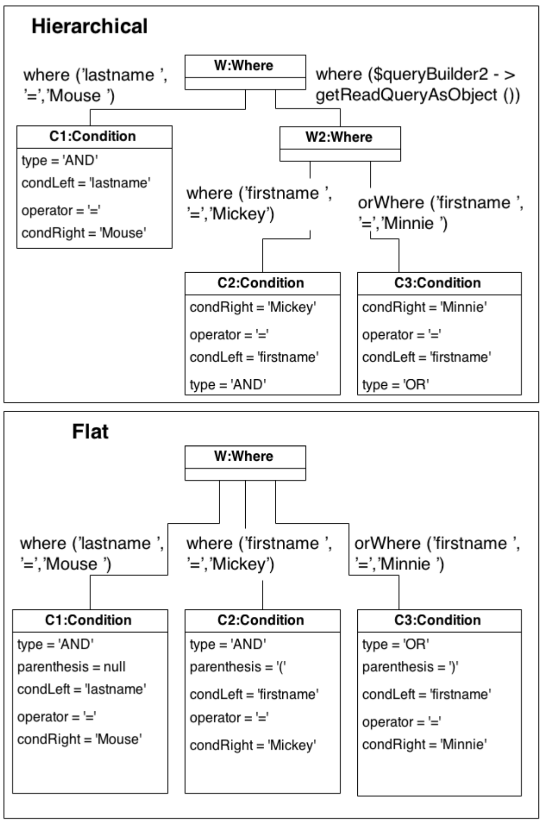


Figure 7: Object diagrams of the hierarchical and flat representation of the query object's where construct

A side effect of referring to a foreign-key in the query is that the MATCH clause needs to reflect the connection between the node or edge, and the *referred element* in a path. In the example in Listing 7 this path is between *user* and *person.* As can be seen from the example, the alias of the referred element will get the foreign-key name and if there are any edges in the pattern then they will get an auto-generated alias.

Pre-processing of Read queries

The pre-processing of aliases and foreign-keys in a read query with only one element in *from* are handled, as in a update or delete query. However, since a read query can include several tables and a single table several times then another solution is needed for the general case. Therefore Bridge-DB relies on the user to make aliases when a table is represented in the query more than once. Furthermore the user must also use the table name or alias of the table when referring to an attribute in BQL. This is similar to SQL's approach and it is done to avoid ambiguity. However, the pre-processing of *where* must still handle foreign-keys and new aliases made during this process.

The *where* part of a read query might also contain a pattern, which needs to be handled. Therefore some prepossessing of the query object must be done to separate a pattern from the other conditions. The conditions which can be translated into Cypher, are defined in EBNF in Listing 8. From the definition of the condition, a pattern can be defined as in Definition 3 and by using this definition the patterns can be removed from the conditions.

```
<condition> ::= <condLeft> <operator> <
    condRight> ;
<condRight> ::= <mixed> | <attribute> | array
    ;
<condLeft> ::= <attribute>;
<operator> ::= 'LIKE' | 'NOT LIKE' | 'IN'
        | 'NOT IN' | '=' | '<' | '>' | '>=' |
                '<=' | '<>' | '!=' | '!<' | '!>'
            ;
<mixed> ::= numeric | string | char;
<attribute> ::= string | <table> , '.',
    string
        | <alias> , '.', string;
<table> ::= string;
<alias> ::= string;
```

Listing 8: EBNF of a condition that can be translated into Cypher

DEFINITION 3. *A condition consisting of condLeft = condRight, is a pattern, iff condLeft and condRight are attributes where one must be a foreign-key attribute and the other, the referenced attribute. This means if condLeft is a foreign-key attribute then its referred element must correspond to the table name and attribute name which condRight refers to, and vice versa.*

An example of both a condition and a pattern is shown in Listing 9. The first *where* clause is a condition even though *u.person_id* is a foreign-key because *condRight* is a numeric value. The second *where* clause is a pattern because *condLeft* is a foreign-key and its referred element is the *id* of *person* which in this case is *p.id.*

```
$queryBuilder->select('u.*')->from('user','u'
    )->from('person','p')->where('u.person_id
    ','=',2)->where('u.person_id','=','p.id')
    ;
```

Listing 9: Example of a pattern and condition

Figure 8 gives an overview of how the pre-processing of a read query is done. The pre-processing action dependents on whether the query contains more than one element in *from.* Provided *from* only contains one element then the alias that has been specified by the user, can be removed and then be handled as described in a delete or update query.

If *from* clause contains more than one table then it is possible that patterns are in the condition, so the conditions and patterns need to be separated. Finally the alias and attribute name in the conditions need to be checked for foreign-key references and replaced by the referred element. Again this might lead to additional patterns and tables in the query. Therefore the additional tables need to be add to *from* and the new patterns to the other patterns. This concludes the pre-processing before doing any translation into Cypher.

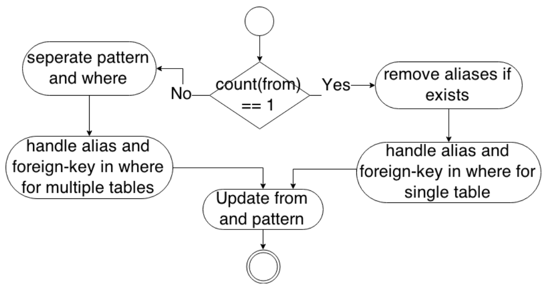

seperate pattern and where
No
count(from) == 1
Yes
remove aliases if exists
handle alias and foreign-key in where for multiple tables
handle alias and foreign-key in where for single table
Update from and pattern

Figure 8: Pre-processing of from and where elements

4.3 Translating Queries into Cypher

When making a Cypher query there are some keywords and notations that should be explained. The typical keywords in a create query are MATCH, CREATE, and RETURN. MATCH finds nodes and edges according to a pattern. CREATE makes the new nodes or edges and RETURN specifies the return values which can be a path, node, or edge with all of its properties, or just a property from a node or edge. An update query also uses the MATCH and RETURN clause, and to update a property it uses the keyword SET. In a delete query which deletes a node and its edges, it uses the keywords MATCH, OPTIONAL MATCH, and DELETE, where MATCH finds the node, OPTIONAL MATCH finds the edges, and DELETE deletes the nodes and edges mentioned. A typical read query also uses the keywords MATCH and RETURN as previously described, but it also uses a WHERE clause to filter the result. In Listing 10 is a small example of a read query in which parentheses symbolize a node and square brackets represent an edge.

```
MATCH (nodeAlias:label1)-[edgeAlias:label2
    ]->(nodeAlias2:label3)
RETURN nodeAlias2
```

Listing 10: Examples of a simple Cypher query

Examples of each CRUD operation are shown in the following sections that describe how each query is translated. It should be noted, that this translator only supports the cascade constraint on foreign-keys.

Create Query

In the BQL *QueryBuilder* a create query consists of a function call to *insert* and one or more to *values.* In Figure 9 is a decision diagram that shows when to make an edge, node, or both. If the table name given in *insert* refers to an edge, then an create edge statement needs to be made, where it needs to find the connecting nodes via the foreign-key properties. If the table name refers to a node without foreign-key properties then a simple create node statement can be made. But if the node contains foreign-key properties then the create statement must create both a single node and an edge for each foreign-key property.

Three examples of translating a create query can be found in Listing 11. The first create query shows how a *person* node which does not have any foreign-key properties, is translated into Cypher. In the second example a *user* node with

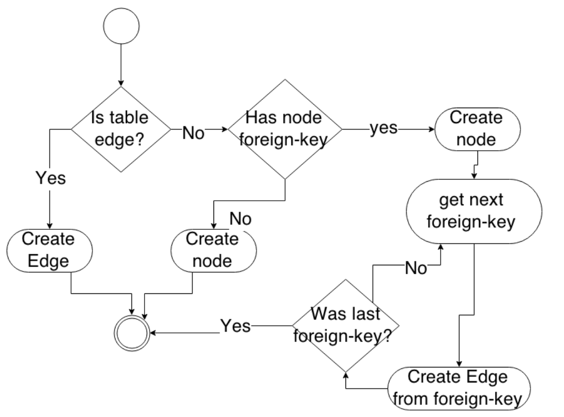


Figure 9: Decision diagram of what create action that should be taken

foreign-key properties is translated to Cypher, resulting in a *user* node and an edge from the *user* to a *person* with the id 5. The last create query in the listing shows how a *friends* edge is created by using *person_id1* and *person_id2* to find the nodes. Notice, that the MATCH clause is different from the read, update and delete queries because a create statement in BQL does not accept any of the *where* conditions, which make this method easier to implement.

```
$queryBuilder->insert('person')->values('id',
    '5')->values('firstname','Mickey')->
    values('lastname','Mouse');
CREATE (n:person{id:5,firstname:'Mickey',
    lastname:'Mouse'} RETURN n

$queryBuilder->insert('user')->values('id',3)
    ->values('username','Mickey')->values('
    password','password')->values('person_id'
    ,5);
MATCH (person_id:person{id : 5}) CREATE (n:
    user{id : 3, username : 'Mickey',
    password : 'password'})-[:person_id]->(
    person_id) RETURN n

$queryBuilder->insert('friends')->values('id'
    ,2)->values('person_id1',3)->values('
    person_id2',5)->values('type','childhood'
    );
MATCH (person_id1:person{id : 3}),(person_id2
    :person{id : 5}) CREATE (person_id1)-[r:
    friends{id:2, type : 'childhood'}]->(
    person_id2) RETURN r
```

Listing 11: Examples of create queries from QueryBuilder to Cypher

Update

The update statement consists of at least two function call *update* and *values* in BQL. Optionally different *where* functions can also be used. An update query has more cases to consider than a create query as can be seen from Figure 10.

If an edge or node is to be updated then the action to be taken, depends on whether a foreign-key property should be changed. So if there are no foreign-key references then it is a simple question of finding the node or edge, and update the relevant properties. However, if the query updates a foreign-key property in an edge then the action depends on whether one of the foreign-key properties is set to *null*. If so, the edge should be deleted, since an edge needs a connection to two nodes to be valid. In the other case, the foreign-key will reference another node, and the edge must be recreated. This means that the current edge needs to be deleted and a new edge with the same or updated properties should be created with the new connecting nodes.

When a node needs to be recreated, it is actually one or several of its edge which was made from a foreign-key property that are recreated. This means an edge might be created, deleted or entirely recreated as described before for remaking an edge.

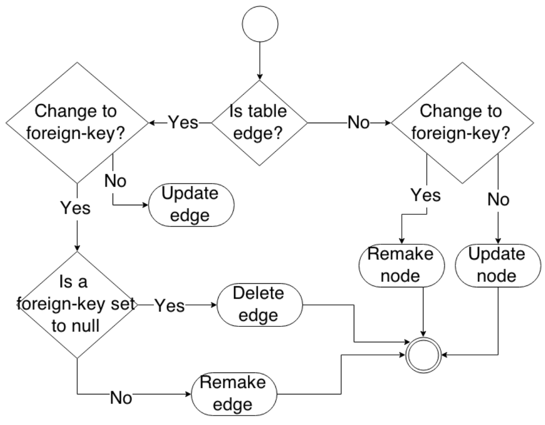


Figure 10: Decision diagram of the update actions that can be taken

In Listing 12 there are two examples of update queries in BQL and their corresponding queries in Cypher. The first is a simple query where all nodes with the label *person* will update its *firstname* property and return the nodes to the user. The second query is simple in BQL, but more advanced in Cypher. In this query a foreign-key property is changed, which means the node is recreated, and this results in a Cypher statement containing MATCH, WHERE, CREATE, SET, WITH, DELETE and RETURN clause. In this query SET copies all the properties from the old edge to the new edge. The WITH clause express that a new query starts and it uses the listed elements from the previous query. So this update query first creates an new edge, copies the properties from the old edge, deletes the old edge, and return the *user* node and the person node it has been connected to.

```
$queryBuilder->update('person')->values('
    firstname','name');
MATCH (person:person) SET person.firstname =
    'name' RETURN person

$queryBuilder->update('user')->values('
    person_id',6);
MATCH (user:user)-[r0:person_id]-(person_id)
    ,(endnode0:person{id : 6}) WHERE user.id
    = 5  CREATE (user)-[newR0:person_id]->(
    endnode0) SET newR0=r0 WITH user,endnode0
    ,r0 DELETE r0 RETURN user,endnode0
```

Listing 12: Examples of an update query in BQL and in Cypher

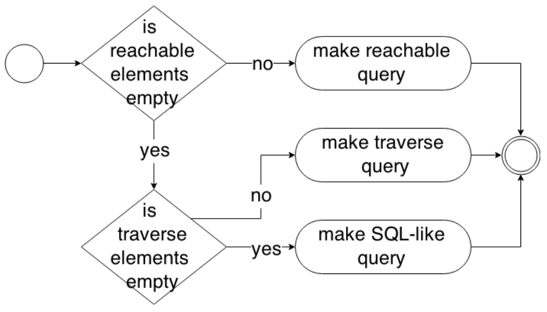


Figure 11: Decision diagram of what type of read query is to be translated

Delete Query

The delete query in BQL is made using the function *delete* and possibly some of the *where* function to filter what should be deleted. Deciding the action in a delete query is simple, if the table name refers to an edge then a delete edge statement should be made. However, if it is a node then the query needs to delete the node and all of its connecting edges.

An example of a delete query in BQL and its corresponding Cypher query is in Listing 13. In the first example a *friends* edge should be deleted which results in a Cypher query containing MATCH, WHERE, and DELETE clauses. The second query deletes a *user* node and its connecting edges. This results in a statement containing a MATCH, WHERE, WITH, OPTIONAL MATCH, and DELETE clause. In this case the *user* node is found by the MATCH and WHERE clause using a *person* node, which is a result of the filtering on the foreign-key property *person_id*. The *user* node is passed on to a new query that finds all of the node's edges, and then deletes both the node and its edges.

```
$queryBuilder->delete('friends')->where('id',
    '=',1);
MATCH ()-[friends:friends]-() WHERE friends.
    id = 1  DELETE friends

$queryBuilder->delete('user')->where('
    person_id','=',2);
MATCH (user:user)-[r0:person_id]-(person_id:
    person) WHERE person_id.id = 2  WITH user
     OPTIONAL MATCH (user)-[r]-() DELETE user
    ,r
```

Listing 13: Examples of a delete query in BQL and in Cypher

Read

A read query can either be a reachable, traversal, or an SQL-like query, see Figure 11. Deciding which type of query it is, is done by checking whether the query contains any reachable elements or traverse elements, and if both of these element are empty then it is an SQL-like query made from a combination of *select*, *from*, and *where*.

The SQL-like read query is more complicated to translate because a query might need several tables. As described in Section 4.2 conditions in *where* has been pre-processed such that all references to foreign-keys are replaced and all patterns are put into their own lists. This makes the conditions ready for translation into the WHERE clause in Cypher. The *select* values also need some extra processing to translate it into the RETURN clause. This is done to remove references to foreign-keys and replace the SQL wild-card * with the relevant properties. The MATCH clause is made from a combination of patterns and tables in *from* which is explained in Appendix D.

In Listing 14 are two examples of an SQL-like read query that are translated into Cypher. In the first query we want to find all *friends* of a *username* and return all data about the friend. In the query there are one condition and three patterns, which are translated into two MATCH patterns. In the second query we want to find a *username* based on a *person_id*. In the BQL query there are no aliases, instead they are created during translation. In addition the query references a foreign-key property that references a *person* which is not represented in the *from* clause and therefore an additional pattern from *user* to *person* is made during translation.

```
$queryBuilder->select('p2.*')->from('user','u
    ')->from('person','p1')->from('friends','
    f')->from('person','p2')->where('p1.id','
    =','u.person_id1')->where('p1.id','=','f.
    person_id1')->where('p2.id','=','f.
    person_id2')->where('u.username','=','
    Mickey');
MATCH  (u:user)-[r1:person_id]->(p1:person),(
    p1:person)-[f:friends]->(p2:person)
    RETURN p2.id, p2.firstname, p2.lastname

$queryBuilder->select('username')->from('
    users')->where('person_id', '=',4);
MATCH (user:user)-[:person_id]->(person:
    person) WHERE person.id=4 RETURN user.
    username
```

Listing 14: Examples of an read query in BQL and in Cypher

To implement *traverse* and *reachable* a basic support for sub-queries is necessary in the Cypher translator. The sub-query supported, is an SQL-like read query that only has a single element in *from*. In Listings 15 and 16 are examples of traverse and reachable queries, respectively. In BQL the queries take two sub-queries as input which both finds a *person* node according to a specified *id*. This is translated into a single sub-query in the Cypher version of the queries which finds both nodes and pass them to the parent query. The parent query finds the shortest path between the two nodes without having any restriction on the path taken. Finally the query returns the result. As can be seen the difference between a reachable and traverse query is the return value which is a number for reachable and a path for traverse.

```
$qb1=$qb1->select('*')->from('person', 'p1')
    ->where('p1.p_id','=',1);
$qb2=$qb2->select('*')->from('person','p2')->
    where('p2.p_id','=',2);

$qb_tra=$qb_tra->traverse($qb1->
    getReadQueryAsObject(), $qb2->
    getReadQueryAsObject());
MATCH (p1:people),(p2:people) WHERE p1.p_id =
     1 AND p2.p_id = 2 WITH p1,p2 MATCH p =
    shortestPath (( p1 ) -[*] -( p2 ) )
    RETURN p
```

Listing 15: Examples of a traverse query in BQL and in Cypher

```
$qb1=$qb1->select('*')->from('person', 'p1')
    ->where('p1.p_id','=',1);
$qb2=$qb2->select('*')->from('person','p2')->
    where('p2.p_id','=',2);

$qb=$qb->reachable($qb1->getReadQueryAsObject
    (), $qb2->getReadQueryAsObject());
MATCH (p1:people),(p2:people) WHERE p1.p_id =
     1 AND p2.p_id = 2 WITH p1,p2 MATCH p =
    shortestPath (( p1 ) -[*] -( p2 ) )
    RETURN COUNT ( p )
```

Listing 16: Example of a reachable query in BQL and in Cypher

5. OPTIMIZER

The optimizer consists of multiple components so to get an overview, we present how the optimizer given a query, are able to find the optimal Query Execution Plan (QEP), see Section 5.1. This only gives a general picture of the components.

In Section 5.2 we presents some algorithm used to gather statistics. This sections is focused on retrieving response times of the queries as well as supplying the cost model with data such that it can create a QEP.

The cost model is described in Section 5.3. The cost model in conjunction with the Credibility Value (CV) compares and selects the most optimal database for a given query. In case the most optimal query execution is not on a single database the cost model uses the results from the plan enumerator, to divide the query and compare derived queries to find a more optimal QEP among the databases as well as the reconstruction of the results to match the original query.

The optimizer focuses on read queries and disregard create, update and delete as these queries cannot be partially executed on one database. The reason for this is these types of queries are supposed to be executed on all databases as we implement full replication across the databases.

5.1 Overview

A sequence diagram of how a query is handled by the optimizer, its main components, and the main actions made during the optimization of the query is shown in Figure 12.

First a user query is sent to the optimizer. During the first step of the *Optimizer*, the initial query is sent to the *Plan enumerator* which makes additional enumeration of the query, such that an QEP can be made for all enumerations of the query. The enumerations are returned to the *Optimizer.*

Next the enumerations are sent to the *Cost model* component which will return a single QEP to the *Optimizer.* In this component, first we retrieve an id of a measurement for each enumeration from *Bridge-DB's internal storage.* This means that if the enumeration has already been executed once then we have previous statistics that the *Cost model* component can use. However, if this retrieval fails then an Measurement Query (MQ) corresponding to this enumeration need to be executed by the *Measurement query* component. After all enumerations have been processed, then the cost model can compare each QEP to find the most optimal and return it to the optimizer.

The QEP can be one or several queries, which should be executed on different databases. Each query is sent to the *Connector* of a database which then will execute the query and return a result to the *Optimzer.* If there were several queries then their results need to be handled by the *Query-post processor* component to join the results into a final result that is returned from the *Optimizer.*

5.2 Statistics gathering

One approach to approximate the post-processing time of a query is to evaluate the data size of different QEPs and use this to select the most optimal plan. However, as this method is great at approximating the QEP with the lowest resource footprint on a database, this is not suitable for Bridge-DB as our focus is on response time.

The reason for measuring the response time on Bridge-DB and not the post-processing time on the individual database servers, is mainly that the size of the returned data may vary. Therefore the bandwidth between Bridge-DB and the database servers will have an impact on the time and our focus is on approximating how much time will pass from the query is sent until a result can be forwarded to the client.

To measure query response time, the original query is received and stored in Bridge-DB's internal data storage. The query is then forwarded to the plan enumerator component. This component is intended to enumerate the query based on its constructs, where a construct is either a select, table reference, or condition. The reason for doing this is to maintain a set of queries derived from the user query including how to reconstruct the original query from the derived queries. By doing this, we are able to execute parts of the query on different databases and let Bridge-DB reconstruct the original query.

Plan enumerator

The queries are enumerated using the algorithm shown in Algorithm 1. The algorithm takes a single query as a parameter. It first checks if there is more than one table. In case there is more than one table, it first creates a new query for each permutation of all possible join combinations and second, creates a new query for each table with all conditions and select statements. Secondly, if there instead are multiple conditions, the algorithm creates one new query for each condition, adds the one table, and adds all select statements to the new query. Thirdly, if neither of these case is in the query then a new query is created with the one table and the select statements, without the conditions. Finally if there are no conditions in the query and no join operations, the algorithm stops and returns the input query. Otherwise for each of the new queries created, the *plan enumerator* algorithm is called with one new query as parameter, creating a recursive function. The algorithm returns all the results from all of the recursive calls of the *plan enumerator.*

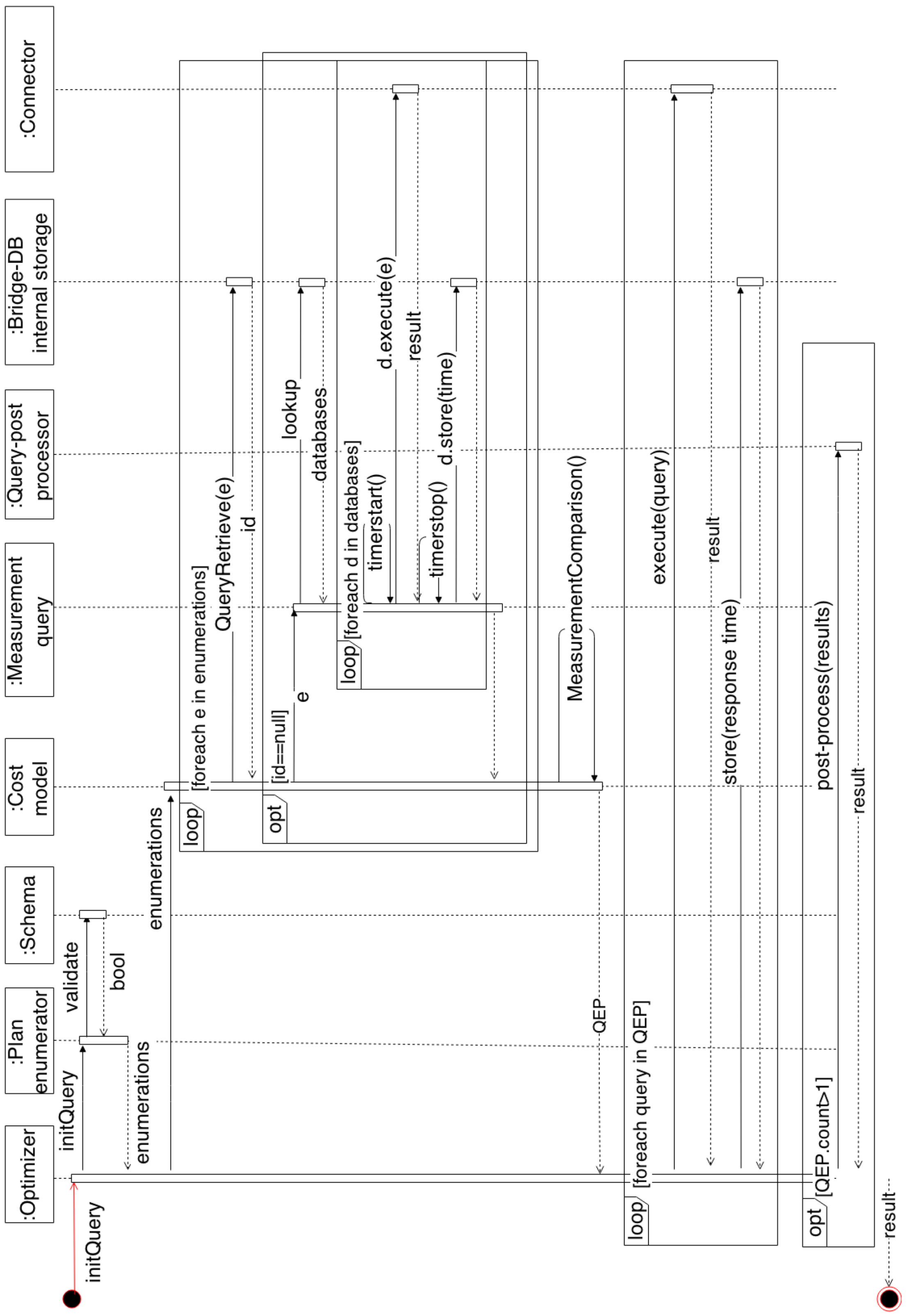


Figure 12: Sequence diagram of the optimizer

```
input  : query
output: Enumerated queries
tmp_q = new query;
tmp_q.add(query.selects);
if count(query.tables) > 1 then
    foreach permutations(query.tables) as permutation do
        new_query = clone tmp_q;
        new_query.add(permutation);
        queries.add(new_query);
    end
    foreach query.tables as table do
        new_query = clone tmp_q;
        new_query.add(table);
        new_query.add(query.conditions);
        queries.add(new_query);
    end
else if count(query.conditions) > 1 then
    foreach query.conditions as condition do
        new_query = clone tmp_q;
        new_query.add(query.tables);
        new_query.add(condition);
        queries.add(new_query);
    end
else if count(query.conditions) == 1 then
    new_query = clone tmp_q;
    new_query.add(query.tables);
    queries.add(new_query);
if count(query.conditions) == 0 && !
count(query.tables) == 1 then
    return query;
else
    foreach queries as query do
        returnQueries.add( queryEnumerator( query ) );
    end
    return returnQueries;
end
```

Algorithm 1: Plan enumerator algorithm

The result from the plan enumerator algorithm is an enumeration of queries, similar to a tree structure. The root of the tree is the initial query and each of the children are derived from the initial query.

An example of this is a query as presented in Listing 17, which contains two join operations, one being a self join. After the execution of the plan enumerator on this query. The result from the plan enumerator can be seen in Listing 18.

```
$q->select("*")->from("user")->from("friends"
    , "f1")->from("friends", "f2")->where("
    user.id" ,"=" ,"f2.user1_id")->where("f1.
    user2_id", "=" ,"f2.user1_id");
```

Listing 17: Initial example query

```
1 $q->select("*")->from("user")->from("friends"
      , "f1");
2 $q->select("*")->from("user")->from("friends"
      , "f2")->where("user.id","=","f2.user1_id
      ");
3 $q->select("*")->from("friends", "f1")->from(
      "friends", "f2")->where("f1.user2_id","="
      ,"f2.user1_id");
4 $q->select("*")->from("user");
5 $q->select("*")->from("friends");
6 $q->select("*")->from("friends");
```

Listing 18: Plan enumeration of example query

In Listing 18 it can be seen that line 1, 2, and 3 are all permutations of the join operations as well as compatible conditions with the referenced tables. Line 4, 5, and 6 are only referencing a single table, and as no condition is compatible with any of those queries, they are removed. This can also be viewed as a tree structure as seen in Figure 13.

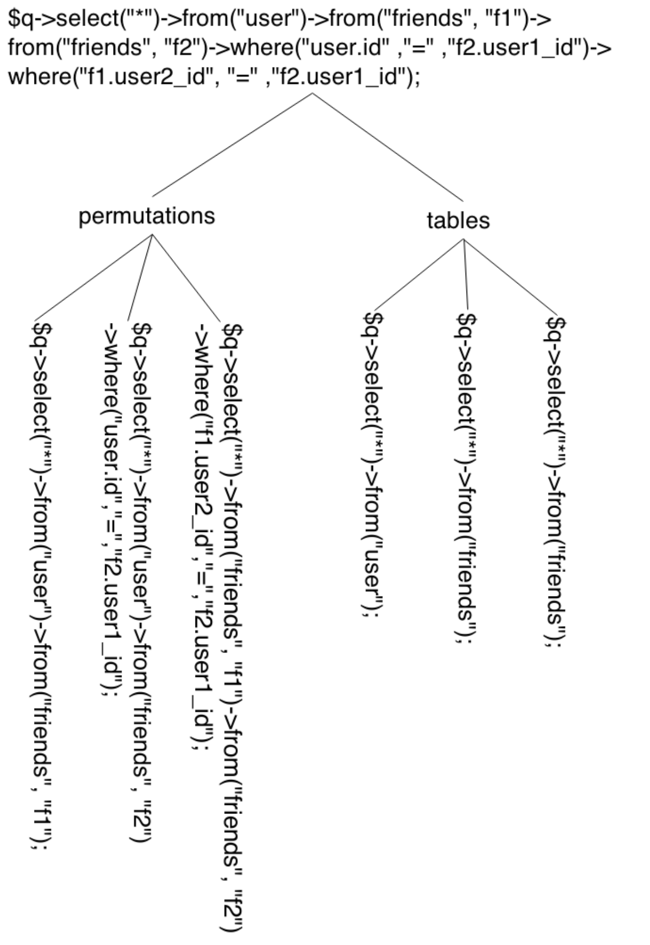

$q->select("*")->from("user")->from("friends", "f1")->from("friends", "f2")->where("user.id" ,"=" ,"f2.user1_id")->where("f1.user2_id", "=" ,"f2.user1_id");
permutations
tables
$q->select("*")->from("user")->from("friends", "f1");
$q->select("*")->from("user")->from("friends", "f2")->where("user.id","=","f2.user1_id");
$q->select("*")->from("friends", "f1")->from("friends", "f2")->where("f1.user2_id","=","f2.user1_id");
$q->select("*")->from("user");
$q->select("*")->from("friends");
$q->select("*")->from("friends");

Figure 13: Enumerated queries

The enumerated queries are used both when we executing MQs where every enumerated query not previously known in Bridge-DB is executed, but also when approximating the most optimal QEP for a user query.

Measurement Queries

As the cost model in Bridge-DB is based on measured response times of the connected databases, the concept of MQs is implemented. An MQ is intended to act as a way of measuring a response time baseline to be used by the cost model when selecting the most optimal QEP and not intended to return an actual result to a user. MQs is mainly visible in bootstrapping queries described in Section 6 but also during execution of user queries in case, the user query is unknown to Bridge-DB. MQs allows the cost model to compare enumerated user queries without the need to execute each enumeration each time they are compared.

To gather statistics the user query is sent as input to Algorithm 1 and the algorithm returns the enumerated queries based on the user query. Each of these queries are then executed one by one where the response time and query result size are stored in the internal storage and marked as an MQ. The reason for storing the result size is, when estimating the execution time for post-processing a query inside Bridge-DB, information about the result size is required. Post-processing of queries is important as the system is designed to execute the queries partially on multiple databases, and then let the post-processor handle the aggregation of the partial results from the individual databases. This is further described in Section 5.3.

Initially, there is no data in the internal database on Bridge-DB about any executed queries and therefore no known measurements. Therefore when Bridge-DB executes a query, it will cause a high load on the databases as the queries are enumerated using the plan enumerator and each query from the plan enumerator is executed as MQs to gather statistics for the cost model.

However, as the response time is stored for each MQ, the MQ will only be executed once. As the enumerated MQs are expected to overlap to some degree, they are considered overlapping subproblems. Therefore we can use principles from Dynamic Programming [1] to speed up the execution of MQs in order to limit the load impact on the database servers.

This is done by hashing each MQ and storing the MQs in the internal storage on Bridge-DB. When the execution of the gatherStatistics algorithm is started, as seen in Algorithm 2, it is considered a batch execution and a batchId value is set. Based on the batchId and the hashed MQ it is possible to retrieve, whether a given MQ has previously been executed and whether that given query was last executed as part of the current batch. In case it was executed in the current batch, the given MQ is an overlapping subproblem and therefore is unnecessary to execute.

An example of this continues with the enumerated queries in Listing 18 which is forwarded to the gatherStatistics algorithm in Algorithm 3. A batchId is created for the gatherStatistics execution and for each query, the internal database is queried to check whether the given query has been executed as part of the current batch. If the query has been executed previously, it is disregarded and will not be executed as an MQ. If it has not been executed previously in the current batch, the query is executed and the response time and result size is stored in the internal database.

Based on the queries in Listing 18, only the queries shown in Listing 19 are actually executed as MQs. As can be seen, one query is missing, which is line 6 in Listing 18. The reason for this is, as mentioned, the initial example query contained a self join on the friends table. As the enumerations contained two identical selections on the friends table, they are considered overlapping sub problems, and therefore the second duplicate query is disregarded when executing the MQs.

```
$q->select("*")->from("user")->from("friends"
    , "f1");
$q->select("*")->from("user")->from("friends"
    , "f2")->where("user.id","=","f2.user1_id
    ");
$q->select("*")->from("friends", "f1")->from(
    "friends", "f2")->where("f1.user2_id","="
    ,"f2.user1_id");
$q->select("*")->from("user");
$q->select("*")->from("friends");
```

Listing 19: Executed MQs

To limit the amount of MQs executed, the retrieve query algorithm in Algorithm 3 simplifies the input query before doing a lookup in the internal database. This simplification is mainly visible in the conditions as they are only considered as referencing an indexed or non-indexed column. The reason for this is, if a column is indexed, the execution time is not expected to change drastically when compared to another indexed column. The same applies for non indexed columns.

A drawback of MQs is, if the database is under load at the time an MQ is executed, the response time results might not reflect the optimal response time which the database is capable of. To limit the impact of this, we implement a method to dynamically adapt to erroneous measurements and changes to the databases called CV.

Credibility Value

The use of CV is intended to let Bridge-DB dynamically adapt to changing database environments. Therefore the CV are assigned to either let MQs or user queries become the dominant factor when calculating the cost of executing the queries. The reason for this design is, ideally the MQs should act as a guideline, but as the database is put under load or the database grows, the user queries supplies updated measurements.

The CV is calculated based on whether it is an MQ or a user query. The CV of an MQ has by default a static value of 10. The reason for choosing a value of 10 is, that we want at least 10 subsequent executions before we depend on the measurements of the subsequent executions as a dominant factor in the cost model. The exact number for queries was found through some experimentation.

The aggregated CV of user queries are calculated according to Definition 4. This calculation is only used when comparing queries and therefore the incoming query can be either the incoming user query or enumerated queries.

```
input  : queryArray (enumeratedQueries)
input  : onlyMissing
batchId = getHighestBatchId();
if batchId > 0 then
    batchId++;
else
    batchId = 1;
end
if isArray(queryArray) == true then
    returnQueries = array();
    foreach queryArray as query do
        if isArray(query) == true then
            gatherStatistics(query);
        else
            foreach internalDatabase.getDatabases() as database do
                storedQuery = retrieveQuery( query, database, batchId);
                if storedQuery['batchId'] != batchId then
                    if ( onlyMissing == true && storedQuery['batchId'] == 0 ) || ( onlyMissing == false ) then
                        startTimer();
                        result = database.query( query );
                        executionTime = stopTimer();
                        internalDatabase.store( storedQuery['queryId'], executionTime);
                    end
                end
            end
        end
    end
end
```

Algorithm 2: Gather statistics

```
input  : query
input  : databases
input  : batchId = 0
output: internal query id
output: batchId
foreach query.conditions as condition do
    if isIndexed(condition) == true then
        tagAndStoreIndexed(condition);
    else
        tagAndStoreUnindexed(condition);
    end
end
selectHash = hash( commaSeperate( sort( query.selects ) ) );
fromHash = hash( commaSeperate( sort( query.froms ) ) );
conditionHash = hash( commaSeperate( merge( sort(indexed conditions), sort(unindexed conditions) ) ) );
query = "select * from queries where selectHash="+selectHash+" AND fromHash="+fromHash+" AND conditionHash="+conditionHash;
if batchId > 0 then
    query = query+" AND batchId="+batchId;
else
    query = query+" AND batchID="+highest(batchId);
end
result = internalDatabase.query(query);
if result then
    return result['primaryKey'],result['batchId'];
else
    query = "insert into queries "+selectHash+", "+fromHash+", "+conditionHash;
    internalDatabase.query(query);
    return primaryKey,0;
end
```

Algorithm 3: Query retrieve algorithm

Definition 4. *Aggregated CV for user queries*
$mq \rightarrow$ the measurement query
$m_i \rightarrow$ the known user query measurements
$n \rightarrow$ the number of known user query measurements
$cv \rightarrow$ the aggregated CV of the subsequent queries

$$cv = \sum_{i=1}^{n} 1 - \left(\frac{|mq - m_i|}{\left(\frac{(mq+m_i)}{2}\right)}\right)$$

The calculation is focused on subtracting 1 from the percentage difference between mq and m_i. Ideally, the user queries should be as close to mq as possible. The result is the aggregated cv of user queries becomes larger than the CV of the MQ as fast as possible making, the user queries the dominant factor. In case the percentage difference is large, a larger set of user queries are required to make them the dominant factor.

5.3 Cost Model

The cost model has its roots in the results from the MQ. The cost model uses these to predict the execution time in Bridge-DB, but the incoming user queries are also used to measure response time. Both the MQ and the user queries are assigned an CV, used to let the cost model progressively learn changes to the database environments, whether the changes are new hardware, new software, or different load on the system. This method is similar to the dynamic cost model, described in Section 2.2.

When a user query is received by Bridge-DB, the cost model retrieves the CV of the MQ corresponding to the user query. It also retrieves the aggregated CVs of all user queries. The cost model selects the dominant CV which is the larger CV of either the MQ or the user queries. Practically this means as long as the aggregated CV of the user queries is smaller than the static CV of MQs, which is 10, Bridge-DB will use the MQ as the dominant measurement and if not, the average of the previously executed user queries are used.

Aside from the CV the cost model can access response times and result sizes of previous user queries and MQs through the internal database. Based on the plan enumerator in

Section 5.2 we have a set of enumerated plans which is forwarded from the optimizer to the cost model. The enumerated queries are structured similar to a tree structure and therefore we reference the queries as nodes.

The first step in the cost model is to traverse the tree structure of enumerated queries. At each traversal step, the CV are recalculated for all types of databases connected to Bridge-DB to choose the dominant measurement. For each node the cost model then compares the dominant measurements of the nodes to find the database with the lowest response time. At each level of the tree, the response time of all siblings are aggregated and the estimated execution of executing local computations is added. The aggregated response time of siblings and local computations is compared with the response time of the common parent of the siblings. The measurement comparison is shown in Algorithm 4 and the estimation of local computations is shown in Algorithm 6.

If the aggregated response time differs with more than 10% from the parent query, the traversal continues as it is expected that further optimization can be done. If not, the traversal stops and if the aggregated response time differs with less than 5%, the parent is used as the optimization achieved in the last step, is not sufficient to defend the additional operations required. These percentages are chosen based on experimentation when executing the tests. The algorithm used to do this is shown in Algorithm 5.

This design can both find a QEP for the execution of a user query using one or multiple databases, by checking the decompositions of the user query against the available databases, the query can be partially executed on the most optimal set of databases.

```
input  : tree
output: result
1  tree.node.aggregateTime = 0;
2  foreach tree.siblings as sibling do
3      tree.node.aggregateTime +=
       sibling.fastestMeasurement();
4      tree.node.aggregateTime +=
       getTimeOfLocalOperation( tree.sibling, tree.node );
5  end
6  tree.node.aggregateTime +=
   tree.node.fastestMeasurement();
7  if percentLower( tree.node.aggregateTime,
   tree.parent.aggregateTime ) > 10 then
8      return measurementComparison(tree.node);
9  else if percentLower( tree.node.aggregateTime,
   tree.parent.aggreTime ) < 5 then
10     return tree.parent;
11 else
12     return tree.node;
```

Algorithm 4: Measurement comparison

The cost model can, based on the measurements and the enumerated queries, select the optimal division of the query in regards to the smallest response time. The algorithm used to do the calculations is shown in Algorithm 5.

```
input  : query
input  : databases
output: result
1  enumerations = planEnumerator(query);
2  if ! internalDatabase.retrieveQuery( query ) then
3      executeMeasurementQuery( query, true );
4  end
5  querySet = measurementComparison(enumerations);
6  if querySet.sublings then
7      resultArray = array();
8      foreach querySet.sublings as subling do
9          resultArray.add(
           subling.getFastestDatabase().execute() );
10     end
11     resultArray.add(
       querySet.getFastestDatabase().execute() );
12     return executeLocalComputation(querySet,
       resultArray);
13 else
14     return querySet.getFastestDatabase().execute();
15 end
```

Algorithm 5: Query selection algorithm

The query selection algorithm returns an QEP as well as computations needed to be executed locally on Bridge-DB to fulfill the user query. At this point, the cost model has calculated a cost for executing the different variations of the user query and the returned QEP is the best in terms of speed, based on the cost model calculations.

The returned QEP is executed by Bridge-DB and in case post-processing is required, meaning that the returned QEP are a decomposition of the user query, the intermediate results from the individual databases are forwarded to the post-processing algorithm. The result of the post-processing algorithm is the final result and it is returned to the client.

Local Cost Calculator

The local cost calculator uses the size of the intermediate results to estimate the execution time of a given operation. As the only two operations that would require post-processing, are a join operation and a condition check. The local cost calculator does not need to take other cases into account. Aside from this, as the join operation is a simple nested-join and the condition check is a loop through all values, the execution times are easily predicted based on only a few preceding measurements of executions.

An example of this is, if we have two tables, shown in Table 2 and 3 and we wish to find the friends of person with $p\_id = 1$, basically two join operations and one condition, as shown in Listing 20.

p_id	**name**	**location**
1	Person name 1	1
2	Person name 2	3
3	Person name 3	2

Table 2: Simplified people table

f_id	person1	person2
1	1	2
2	2	3

Table 3: Simplified friends table

```
$q->select('*')
->from('people','p1')->from('people','p2')
->from('friends')
->where('p1.p_id','=','friends.person1')
->where('friends.person2','=','p2.p_id')
->where('p1.p_id','=','1')
```

Listing 20: Example query in BQL

Bridge-DB first checks conditions and then join operations. Therefore, the algorithm shown in Algorithm 6 receives as input *condition* as operation parameter and the *data* parameter is an array which contains the size of the table. The local cost calculator executes a query in the internal database for the last 10 previous measurements for the given operation, in this case the 'condition' operation. Each of these measurements consist of execution time and the number of rows.

As the implementation of the local post-processor is a simple loop through the rows, we simply divide the the time with the table size for each 10 previous measurements and calculate the average. This will give us an approximation of the time needed to check the condition of one row. This approximation is multiplied with the number of rows for the operation of which the cost is calculated and the result is returned.

In the next step, the cost of join operations are calculated. They are calculated one by one, but the same procedure is used in both cases. When calling the local cost calculator, the operation parameter is set to *join* and the *data* parameter is an array with two values, one being the number of rows in the first table, and the second value is the number of rows in the second table. The algorithm retrieves the previous 10 measurements for the *join* operation where each measurement consist of the number of rows resulted from a cartesian product and the execution time. The reason for using the number of rows resulted from a cartesian product is because the join operation is a simple nested join implementation.

The execution time is divided by the number of rows and the average is calculated over the 10 measurements. The result is then multiplied with the number of rows resulted from a cartesian product of the two input tables and the result is returned.

This solution depends highly on the same principles the general cost model does which is the MQs as the previous measurements dictate the approximated execution time.

6. MEASUREMENT BOOTSTRAPPING

As executing MQs can be expensive, we limit this problem through bootstrapping Bridge-DB by executing the most basic MQs. As we declare the schema locally in Bridge-DB we

```
input  : operation
input  : data = array()
output: time estimate
1  timedResults = internalDatabase.query("SELECT
   time,dataStatistics FROM localOperations WHERE
   operation="+operation+"" ORDER BY id DESC
   LIMIT 10);
2  storedAverages = array();
3  if operation == "condition" then
4      foreach timedResults as timedResult do
5          tableSize =
           timedResults['dataStatistics']['tableSize'];
6          averageTimePrRow =
           timedResult['time']/tableSize;
7          storedAverages.add(averageTimePrRow);
8      end
9      return data['tableSize'] / average(storedAverages);
10 else if operation == "join" then
11     foreach timedResults as timedResult do
12         table1Size =
           timedResults['dataStatistics']['table1Size'];
13         table2Size =
           timedResults['dataStatistics']['table2Size'];
14         averageTimePrRow =
           timedResult['time']/(table1Size * table2Size);
15         storedAverages.add(averageTimePrRow);
16     end
17     return (data['table1Size']*data['table2Size']) /
       average(storedAverages);
```

Algorithm 6: Local cost calculator

gain the ability to pre-create these MQs. This method is similar to the black box cost model in Section 2.2. The algorithm used to do this is shown in algorithm 7.

A subset of the queries generated from Algorithm 7 can be seen in Listing 21 and Listing 22 for SQL and Cypher respectively where *COLUMN* and *VALUE* is used as general placeholders. *COLUMN* represent each column in the given table and *VALUE* represents a random value in the given *COLUMN*.

```
SELECT 1
SELECT * FROM user
SELECT * FROM friends
SELECT * FROM user WHERE COLUMN=VALUE
SELECT * FROM friends WHERE COLUMN=VALUE
...
```

Listing 21: Subset of autogenerated MQs

```
RETURN 1
MATCH(n:user) RETURN n
MATCH(n:friends) RETURN n
MATCH(n:user) WHERE COLUMN=VALUE RETURN n
MATCH(n:friends) WHERE COLUMN=VALUE RETURN n
...
```

Listing 22: Subset of autogenerated MQs

The first line in both listings, we do a selection on a static value, which is used to measure the response time

```
output: Array of bootstrapping queries
sqlQueries.add("SELECT 1");
cypherQueries.add("RETURN 1");
tableNames = declaredSchema.getTableNames();
foreach tableNames as tableName do
    sqlQueries.add("SELECT * FROM tableName");
    cypherQueries.add("MATCH(n: tableName) RETURN n");
    foreach tableNames.column as column do
        value = getRandomValue(tableName, column);
        sqlQueries.add("SELECT * FROM tableName WHERE "+column+"='"+value+"'");
        sqlQueries.add("MATCH(n: tablename) WHERE n."+column+"="+value+" RETURN n");
    end
end
return sqlQueries, cypherQueries;
```

Algorithm 7: Measurement Query auto generation

of the database without retrieving any actual data from the database. In the rest of the queries, a *SELECT ** is used for each query. This is to retrieve the largest possible result, and through that, indrectly test the available bandwidth of the individual databases. Lastly, based on the selection on each table, we extract a random value from each column of each table and do a *SELECT ** with the column and random value as a condition. This tests each database' ability to handle conditions.

By executing these pre-created MQs we limit the number of MQs Bridge-DB has to execute while processing user queries. Aside from this, by bootstrapping Bridge-DB we also warm up the cache on the connected databases.

7. EVALUATION

In this section we evaluate the overhead, response time, and bandwidth consumption of Bridge-DB using one setup. We also test the ability of the optimizer to change its QEP of a query when the run-time behaviour of the databases changes.

7.1 Test hardware

During testing we use 4 identical servers, each with an Intel Q9440[12] processor, 8 GB memory, one 250 GB Samsung 840 Evo[19] harddrive. Each server is connected to a gigabit network which is isolated from other networks to avoid network interference. The operating system used is Ubuntu 15.04[23]. To ensure a fair comparison, each server is dedicated to a single part of the system each; Client, Bridge-DB, PostgreSQL and Neo4J.

7.2 Datasets

When evaluating Bridge-DB, we use three datasets. The first is a dataset with a bias towards graph operations and therefore Neo4J should have an advantage. The second is a dataset with a bias towards relational databases which we have adapted from the TPC-C dataset [22], and PostgreSQL should have the advantage. The final dataset is a combination of the two aforementioned datasets.

Social Graph Dataset

Renzo Angles et al. [6] present the Graph Data Benchmark (GDBench) which evaluates the performance of a graph database based on a social-network dataset. The data is created by the data generator (GDGenerator) which is included in their benchmark. To test a graph dataset against Bridge-DB we have used a dataset generated by GDGenerator.

The dataset is shown in an ER-diagram in Figure 14. It consists of four elements *people*, *webpages*, *friends*, and *likes*. In the graph database this will result in the nodes *people* and *webpages*, as well as the edges *friends* and *likes*. In the relational database *people* and *webpages* are data tables while the others are join tables.

The queries executed on the dataset are listed below and they are intended to reflect a few queries common to a social network. These queries are a subset of the queries in GDBench, and the remaining queries in the benchmark cannot be expressed in BQL.

Query 1 Find a person with a given name

Query 2 Find all people who likes a given webpage

Query 3 Find all webpages a person likes

Query 4 Find a person name from person id

Query 5 Check if there is a path between person 1 and person 2

Query 6 Find a path between person 1 and person 2

Query 7 Find a common friend between two people

Query 8 Find all webpages two people both like

Query 9 Find all friends of friends of a person

Query 10 Find all webpages liked by friends of friends

Query 11 Find all people who likes a webpage which person with id 18 likes

Queries 5 and 6 are a reachable and traverse query, respectively, while the others are SQL-like queries. When we have an SQL-like query which contains a *friend* relationship, then there is a small problem finding all friends. This is due to the undirected nature of the relationship which cannot be expressed in BQL, which uses a directed approach. So to get all friends, BQL would need a construct like a union to express the correct query. Therefore we have a directional constraint on queries 7, 9, and 10 which gives a subset of the desired result, and this applies to both databases so we will get the same result. For query 7 this actually means that it finds a common friend between person 1 and person 2, where the relationship has been initialized by persons 1 and 2. All queries have been written via the *QueryBuilder* which can be found in Appendix F.

The size of the dataset is 362 MB in PostgreSQL and 1.09 GB in Neo4J. The node and edge statistics can be found in Table 4.

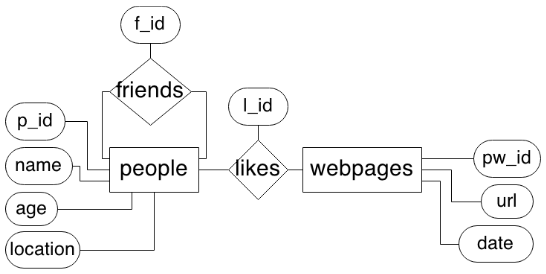


Figure 14: An ER-diagram of the social graph dataset used for testing

Type	Label/Table name	No. of elements
Node/ Data table	people	130.000
	webpages	70.000
Edge/ Join table	friends	1.533.881
	likes	1.545.009

Table 4: Statistics of the social graph dataset

Relational Dataset

To test Bridge-DB against a relational dataset, we have chosen to use the dataset of the TPC-C Benchmark [22]. HammerDB [11] has made a data generation and loading tool which we have used to generate the data.

An ER-diagram of the data set is presented in Figure 15. The attributes of tables in the dataset are not represented in ER-diagram. In [22] the dataset has been specified, but we have made some changes because Bridge-DB does not support all of the design constructs of the TPC-C dataset. There have been made two modifications. First each table must have a single column that can uniquely identify a tuple in the table, this means that all tables which have a composite primary key or no primary key, get a surrogate primary key. This change affects all but *item* and *warehouse*. The second change is that each foreign key may only reference one column in a table. In the original TPC-C dataset a foreign-key can reference multiple columns which is the composite primary key of the table. Instead, each of those foreign-keys is removed and replaced with a foreign-key that references the new primary key from the first change.

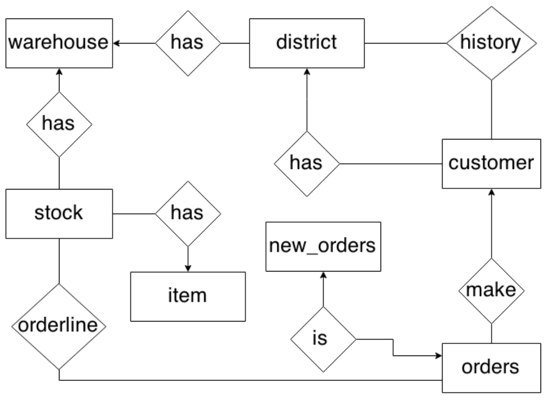


Figure 15: An ER-diagram of the relational dataset used for testing

The queries executed on the relational dataset, are listed below and they can also be found in Appendix F. We have written these queries to reflect some of the common datawarehousing queries which also can be expressed in BQL.

Query 1 Find a customer with id 90032

Query 2 Find all customers in district 3

Query 3 Find all items a customer with id 90032 have bought

Query 4 Find all overlapping items two customers have bought

Query 5 List all customers who have bought something from warehouse 3

Query 6 List all items available at warehouse 3

The size of the dataset is 414 MB in PostgreSQL and 2.28 GB in Neo4J. The statistics of the dataset can be found in Table 5.

Type	**Label/Table name**	**No. of elements**
Node/ Data table	warehouse	4
	customer	120000
	district	40
	item	100000
	new_order	36000
	orders	120000
	stock	400000
Edge/ Join table	history	120000
	order_line	1199766

Table 5: Statistics of the relational dataset

The Merged Dataset

This dataset is a combination of the graph and relational datasets. We have used the relational dataset as base and then connected the *friends* relationship to *customer*, and the *like* relationship has been added between *customer* and *item*. This result in a dataset as presented in Figure 16.

We used GDGenerator to generate a new set of *friends* and *likes* relationship which correspond to the number of elements found in *customer* and *item*. To do this we have made a one-to-one mapping between *people* and *customer*, as well as *webpages* and *items*, see Definitions 5 and 6.

Definition 5. *People-To-Customer Mapping: f(x)=x*

Definition 6. *Webpages-To-Item Mapping: f(x)=x - no. of people*

The queries executed on the merged dataset are listed below and the BQL versions can be found in Appendix F. The reservations stated about the queries in the social graph dataset also apply to these queries.

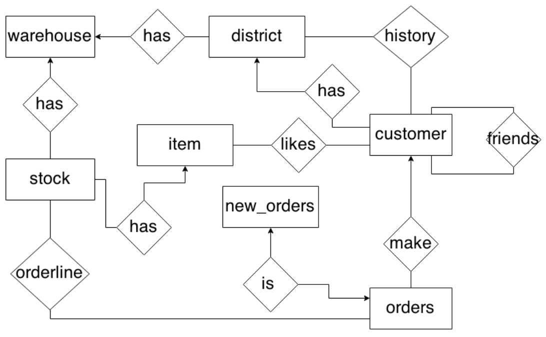


Figure 16: An ER-diagram of the merged dataset used for testing

Query 1 Find all friends of friends who have bought the same product

Query 2 Find all friends of a customer in the same district

Query 3 Find all friends who have liked a given item

Query 4 Find all friends who have both liked and bought a given item

Query 5 Find all items a customer have both liked and bought

The size of the dataset is 2.1 GB in PostgreSQL and 3.32 GB in Neo4J. The statistics of the dataset can be found in Table 6.

Type	Label/table name	No. of elements
Node/ Data table	warehouse	4
	customer	120000
	district	40
	item	100000
	new_order	36000
	orders	120000
	stock	400000
Edge/ Join table	history	120000
	order_line	1199766
	likes	1417423
	friends	1406550

Table 6: Statistics of the merged dataset

7.3 Test Execution

The measurements of response time have been done on the client-side. Meaning, the timer is started when the query is sent using the Bridge-DB client library, and stopped when the entire result has been received. The result is asserted against a known correct response. If the assertion is true then the timed execution is stored, otherwise the test would have failed. The assertion has been done to avoid erroneous database results from contaminating the test results. Each of the queries is executed 50 times and the average execution time is stored and presented here as a result.

The test executions are split into three parts in which the first two executions are not using the optimizer. The first is a cached result meaning the database has been warmed up with a query, and followed by the same query being executed 50 times. Then the average response time is calculated and used in the results. This test demonstrates how the database can perform at its best. The second execution is a non-cached result meaning caching is disabled in the operating system and before each execution, the database is rebooted to clear any active cache. Both of these execution have been done on both PostgreSQL and Neo4J through Bridge-DB.

The third and last execution is with an enabled optimizer in Bridge-DB. As the cost model automatically executes measurement queries, it will automatically warm up the cache. Therefore the results can both be partly cached and partly non-cached results. The queries can also be executed entirely on one of the database or using both databases, depending on the optimizer estimates.

7.4 Overhead Result

When executing the tests we measured the overhead of the cost model both in execution of user queries, but also the execution of an MQ.

The general overhead of the optimizer is on average 8 ms calculated by starting a timer when the query is received and stopping the timer when the QEP is ready for execution. This process consists of enumerating the queries, look-up in the local database for known measurements, comparison of the enumerated queries to find a QEP as well as query translation.

Secondly, the overhead of executing MQs consists of both response time and data size. In terms of response times, Bridge-DB has an overhead of 21%. This is measured by timing every time Bridge-DB executes a query, no matter whether it is a user query or an MQ and stores the time. Additionally, before each MQ a seperate timer is started, timing only the execution time of the MQ. By calculating the percentage difference between the total time spent on executing queries and the time spent on execution MQs we get the overhead in terms of response time. In terms of data size, the aggregated transmitted data of executing the measurement queries is 14 GB, so multiple times the full size of each database.

Out of the 21% overhead in terms of response time, 96% is during bootstrapping of the system and the last 4% is during regular use when queries not previously executed needs to be measured. Out of the 14 GB of transmitted data when executing MQs, less than 100 MB is during regular use and the rest is during bootstrapping.

7.5 Query Response Time Results

In this section we will present our results when executing our tests. We both present the response times for each dataset and how the optimizer has treated queries which was enumerated.

Social Graph Dataset

The test results of social graph dataset are shown in Figure 17. As can be concluded from the figure, there is a significant difference between executing the queries on Neo4J when the cache is enabled or disabled. The average percentage difference is 186,7%, demonstrating the efficiency of the built-in cache in Neo4J. On PostgreSQL this percentage is 32,54%, which demonstrates that PostgreSQL also obtains a significant speedup over time, but still not as significant as Neo4J.

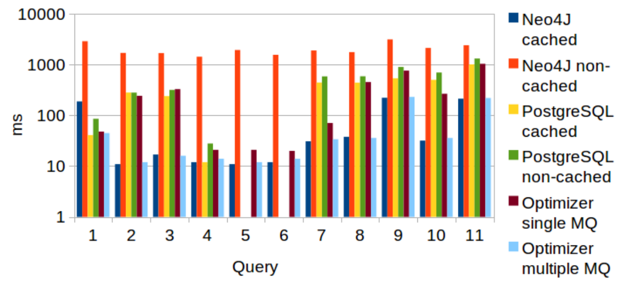


Figure 17: Query responds times in the social graph dataset

When comparing the response times with enabled cache on both Neo4J and PostgreSQL, the average percentage difference is 137,85%, in favor of Neo4J showing that over time, Neo4J in our test cases becomes the fastest database for most of the queries. This changes when comparing the non-cached response times on both Neo4J and PostgreSQL with a percentage difference of 117,32% in favor of PostgreSQL.

When enabling the cost model and executing the same set of queries, the results are more similar to those of the cached results. When examining which database the queries have been executed on, see Table 7, then we found that queries 1-4, 8, and 11 is run on PostgreSQL, queries 5 and 6 run on Neo4J while query 7, 9 and 10 is decomposed and uses both databases to execute the query. It should be noted that queries 5 and 6 have not been executed on PostgreSQL since the SQL translator does not support recursive queries, but as we have shown in our previous work[8] Neo4J would still outperform PostgreSQL.

When testing how the cost model selects QEPs when execution an MQ multiple times, we discovered that all queries but query 1 was executed on Neo4J and only query 1 was executed on PostgreSQL. The reason for this is, the second MQ only get cached results from both databases, and on Neo4J this is a simple lookup in a key-value store for the query and its result and therefore not an accurate measurement.

Query	Execution type
Query 1	All on PostgreSQL
Query 2	All on PostgreSQL
Query 3	All on PostgreSQL
Query 4	All on PostgreSQL
Query 5	All on Neo4J
Query 6	All on Neo4J
Query 7	Decomposed
Query 8	All on PostgreSQL
Query 9	Decomposed
Query 10	Decomposed
Query 11	All on PostgreSQL

Table 7: Execution of social graph dataset's queries

It is notable that the cost model does not choose Neo4J more often when we look at the cached performance of Neo4J. This suggests that the cost model favors PostgreSQL, which actually only beats Neo4J's cache result in query 1 which is a selection on a non-indexed value. However, it also shows how the cost model improves the response time of query 7 and 10 compared to the cached PostgreSQL results. Query 9 which also is a decomposed query does not perform better than the cached result from PostgreSQL.

Through analysis of the decisions made by the cost model, the favoritism of PostgreSQL is based on the initial MQ executed. This makes the cost model select PostgreSQL as the desired database when the optimizer finds, that the optimal QEP is to execute the initial query. Because of this, Bridge-DB does not test Neo4J again, unless PostgreSQL becomes slower than the initial MQ executed on Neo4J. A solution to this problem is to re-execute the **mqs!** (**mqs!**) after an amount of user queries. This is also diskussed further in Section 9.

When looking closer into how query 7 is decomposed then we observe that the query is decomposed into three composite queries, see Figure 18. First we have two queries that finds two people from an id which both are executed on PostgreSQL, and one query that joins all the tables together, returns it to the optimizer which then joins the results from the databases together to find the final result. This is also the general description of the QEP for query 9 and 10, which can be found in Appendix H. This result also suggests that we have much traffic between Neo4J server and the Bridge-DB server. This has also been tested and the results are discussed in Section 7.6.

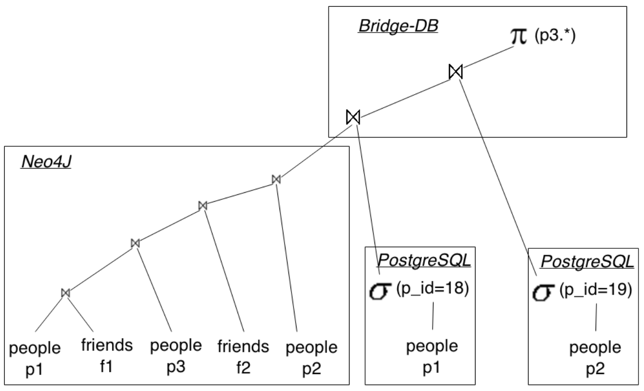


Figure 18: High-level description of QEP of query 7

Relational Dataset

The results of the relational dataset are shown in Figure 19. Some of the same patterns appear as with the social graph dataset. Again Neo4J is able to obtain a much lower response time when the cache is enabled, which results in a percentage difference being 83,57%. For PostgreSQL the percentage difference has decreased to only 4,2%.

When comparing the cached response times of Neo4J and PostgreSQL, the percentage difference is 120,85% in favor of Neo4J and when comparing the non cached response times, it repeats with a percentage difference of 53,97% in favor of

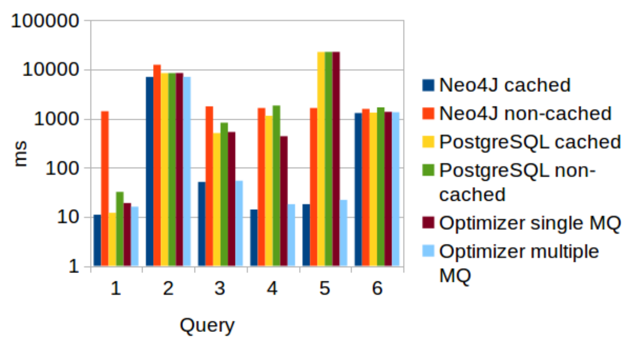


Figure 19: Query responds times in the relational dataset

PostgreSQL.

When enabling the optimizer we also found similar results as for the social graph dataset. The response time tends to follow the cached results from PostgreSQL and all queries but one is also solely executed on PostgreSQL as shown in Table 8. As can be seen from Figure 20 only query 4 has been divided into composite queries in a similar way to query 7 in the social graph dataset.

When executing the MQs multiple times, all queries are executed solely on Neo4J, again accredited to Neo4Js cache.

Query	Execution type
Query 1	All on PostgreSQL
Query 2	All on PostgreSQL
Query 3	All on PostgreSQL
Query 4	Decomposed
Query 5	All on PostgreSQL
Query 6	All on PostgreSQL

Table 8: Execution of relational dataset's queries

Merged dataset

The test results of the merged dataset are shown in Figure 21. Again we can observe that Neo4J outperforms PostgreSQL when looking at the cached results as well as PostgreSQL performing better than Neo4J when the cache is disabled.

The response time difference between the cached executions is 175,57% in favor of Neo4J and the response time difference between the non-cached executions is 102,5% in favor of PostgreSQL.

When enabling the cost model we find that query 2 is executed on PostgreSQL, but the other 4 are divided into composite queries and executed on either PostgreSQL or Neo4J, see Table 9. From Figure 22 it is apparent that the QEP of query 1 is similar to the two previous example. The query execution plans for 3-5 can be found in Appendix H, but they show the same pattern.

When executing the MQs multiple times, all queries are again executed on Neo4J.

Query	Execution type
Query 1	Decomposed
Query 2	All on PostgreSQL
Query 3	Decomposed
Query 4	Decomposed
Query 5	Decomposed

Table 9: Execution of merged dataset's queries

7.6 Data Traffic Results

As mentioned in Section 7.5 by looking at the QEP of the decomposed queries we would find that the data transfer between the Neo4J server and Bridge-DB server to be high. In Figures 23, 24 and 25 are shown the data size returned from Neo4J, PostgreSQL, and Bridge-DB for each of the datasets. It should be noted that the column for query 5 in Figure 23 is too small to be seen.

As we expected the data size returned from Neo4J for each of the decomposed queries, are significantly higher than final results returned from Bridge-DB especially query 4 from the relational dataset in Figure 24.

This data traffic is a significant disadvantage of the optimizer in this test setup. However, due to the dynamic nature of the cost model this might not become a problem with a different setup. For example the network connection can be slower, so the cost model will find another QEP which does not use as much data transfer, however, this might also lead to the optimizer choosing only either PostgreSQL or Neo4J. In another example the transferred data might increase to a point where the optimizer finds a QEP with a lower cost.

7.7 The Run-time Behaviour of the Optimizer

In this test we want to test the cost model's ability to dynamically choose an QEP while each of the external databases is under a different load. We use the same queries on the three datasets, each query is executed 200 times, and the external database servers is under high load at different iteration interval. The additional load on the database increases the response time with several 1000 ms.

The results of the test is presented in Table 10 which both presents when database is under load and when the optimizer chooses a different QEP. As can be seen from the table first we run 50 iteration without any load on the external database servers such that we would get the same results as in the response time tests. After 50 iterations we put PostgreSQL's server under load which after 14 additional iterations causes the optimizer to choose another QEP such that all queries are executed on Neo4J. After 100 iterations we switch the load from PostgreSQL's server to Neo4J's server and then after 14 iterations the optimizer chooses PostgreSQL to execute the queries on. This behavior continues until the we stop the test after 200 iterations.

From this test we can conclude that the optimizer is able to change QEP at run-time based on response times that is collected when a query has been executed. We can also conclude when a database is under a high load, it takes 14 iterations to change the QEP for a query.

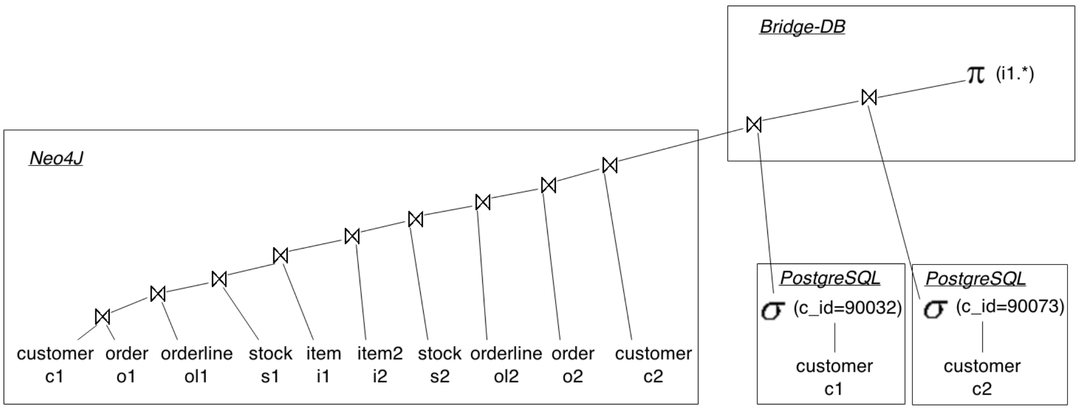


Figure 20: High-level description of QEP of query 4

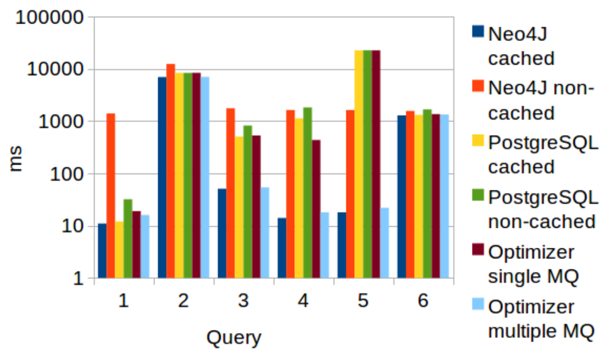


Figure 21: Query responds times in the merged dataset

Iteration	Action
1	start, no database is loaded
50	PostgreSQL is under load
64	Optimizer only chooses Neo4J
100	Neo4J under load, PostgreSQL not under load
114	Optimizer only chooses PostgreSQL
150	PostgreSQL under load, Neo4J not under load
164	Optimizer only chooses Neo4J
200	Stop test

Table 10: Results from testing with changing conditions for the database which exposes the dynamic nature of the cost-model

7.8 Test Conclusion

Even though Neo4J demonstrated an exceptional ability to execute queries fast when the query has previously been executed, it will not suffice in a real world scenario. PostgreSQL demonstrates an ability to execute the queries faster than Neo4J when the result is not cached and even though the result is cached, the execution time is still good.

When enabling the cost model, we can see that in several cases, we gain a speedup in terms of execution time compared which closely resembles the cached results, even though the query at the execution time is unknown and therefore is not cached. This is achieved through the decomposition of the queries as well as the use of measurement queries. Each time a new query is received, then all of the derived queries have already been executed on all databases causing a warm up of all database caches, not necessarly of the entire query but parts of it.

When queries are executed, they showed a similar pattern across the different tests. They are consistently decomposed when self joins are executed as the join operation is consistently executed on Neo4J. Also conditions are consistently executed on PostgreSQL and Bridge-DB only handles the merging of the database results. This also has resulted in a large amount of data traffic between the Bridge-DB server and Neo4J server.

When alternating load on the databases, we can see that Bridge-DB is able to correctly switch between databases and therefore is able to dynamically handle changes to the database environment.

When testing whether it makes a difference for Bridge-DB to execute the MQs multiple times in order to have the cached response times used as measurements, we discovered that Bridge-DB begins consistently using Neo4J to execute its queries except in test 1 query 1. This is due to Neo4Js aggressive caching methods, which causes the measurements to be more related to that of a lookup in a small key-value store then that of executing queries on a graph database.

The results retrieved by measuring the overhead of using this cost model can to some extent be surprising. When executing the measurement queries, the amount of transferred data exceeds the actual size of the databases by several times their actual size. This does not immediately impact the actual performance of the system as most of this data transfer is done during the bootstrapping of Bridge-DB and not during actual use. Regarding the high overhead in terms of response time, the same thing applies that most of this time is spent when bootstrapping Bridge-DB.

An additional test has been executed to test the scalability

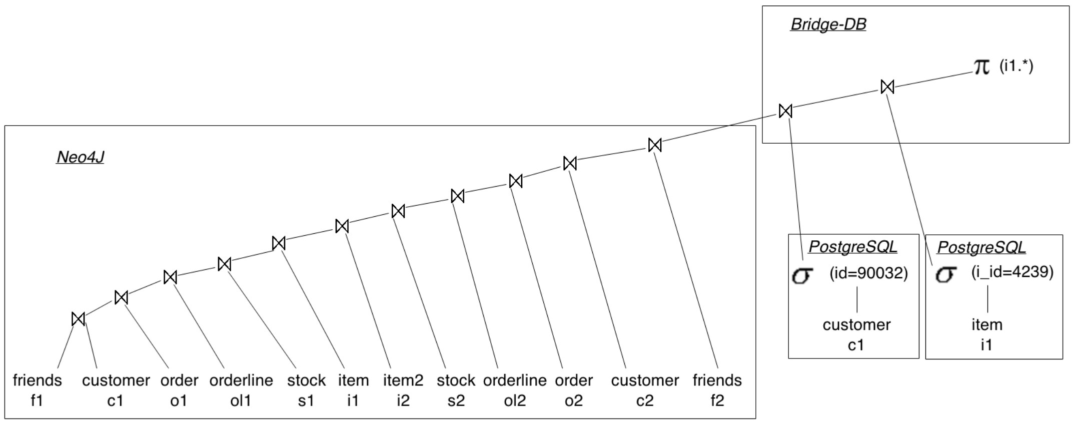


Figure 22: High-level description of QEP of query 1

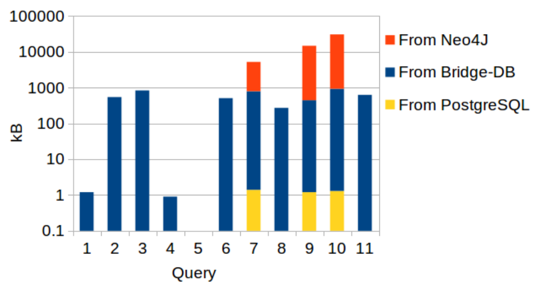


Figure 23: Data sizes for the graph dataset

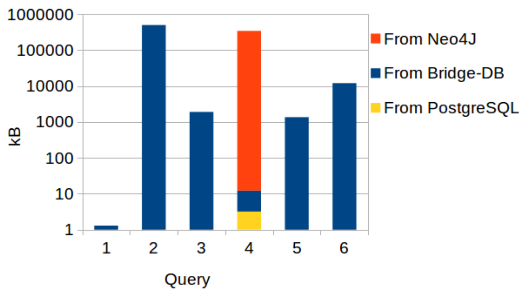


Figure 24: Data sizes for the relational dataset

of the system by starving the databases of available memory. The test result from the old version of Bridge-DB [8] demonstrated that Neo4J did not perform well if it was constrained by a low amount of available memory. But as this result could not be replicated in this environment due to the use of solid state drives, giving Neo4J access to a swap partition capable of serving Neo4J fast enough. This test result is not shown.

8. CONCLUSION

We have presented a method for implementing a distributed database system, which is called Bridge-DB, between a client and two heterogeneous databases, capable of decomposing and executing queries on multiple databases with the purpose of speeding up response times by allowing the databases to work in unison.

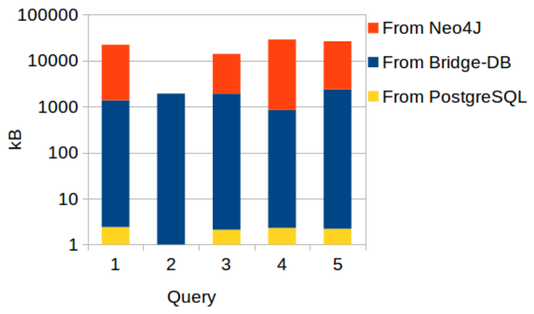


Figure 25: Data sizes for the mixed dataset

To do this, we proposed an architecture consisting of a TCP-socket based communication protocol, connected to an optimizer component capable of estimating the most optimal division of queries based on the suggestions from a cost model.

The cost model implements concepts from both dynamic cost models and black box cost models to both have initial measurements to base decisions upon, but also continuous measurement of query response times.

We also proposed a method to declare a global schema, which can be imposed onto the connected databases to ensure schema consistency across all databases. The schema is also used for query translation and validation.

Furthermore, we demonstrated a query language called BQL with all CRUD operations implemented, and designed to be translated into both Cypher and SQL.

Our results have shown that the use of a distributed database system can be feasible and is able to improve the response times of queries and achieve response times close to those of cached responses.

One drawback with the current method is the high bandwidth usage internally caused by Neo4J consistently handling all join operations when a query is decomposed. This caused a large result being returned to Bridge-DB for further processing.

9. FUTURE WORK

Referential integrity on Neo4J

In BSL we can set the referential integrity on a foreign-key, but is not comparable with Neo4J referential integrity. This needs to be handled by Bridge-DB, which currently only supports a cascading referential integrity constraint. This means when we want to delete or update a node then Bridge-DB needs to find the connecting nodes, check whether the action is legal, and whether some adjustment needs to be made to the connecting nodes, and afterward performs the necessary actions.

Additional Support for Expressing Patterns

During testing we found that it was hard to express all the join patterns using our relational approach, when we want to find the friends of a friend. Therefore it would be better if BQL could be extended further, such that patterns could be expressed more simply, where the schema then supplied the additional information when translating the query to Cypher or SQL.

Adding Sub-query Support in Cypher Translation

The Cypher translator can only translate some simple sub-queries into Cypher and this is only *traverse* and *reachable* queries. This should be extended such that sub-queries also are possible in SQL-like queries. Currently the supported sub-queries are only allowed to include a single table so this should also be extended to include more complex queries.

Partial-Replication across the Databases

Currently Bridge-DB use full replication across the databases such that all data are in both Neo4J and PostgreSQL, but it would be an advantage if a partial replication method was used instead. For example PostgreSQL is good at store and retrieve data while Neo4J is good at relationship and traversing through them. This would also result in less data transferal between Bridge-DB's server and Neo4J's server which was a problem with Bridge-DB's cost model.

Consistency Control between Databases

As complexity is an disadvantage of using multiple databases so is the lack of support for data consistency between the databases. In Bridge-DB it is possible to extend the concept to include some data consistency control between the external database. This could be to insure Atomicity, Consistency, Isolation and Durability (ACID) or Basically Available, Soft state, Eventually consistent (BASE) properties.

Better database selection by the cost model

During the testing phase, it was revealed that the cost model would quickly create a bias towards either database in some use cases. This is caused by relying too much on the initial MQ causing all subsequent queries to be executed on the database which initially demonstrated the best response times.

The cost model would only change database if the selected database became slower than the initial executed MQ on another database. By solving this problem, Bridge-DB would be better suited to adapt to changing environments, whether it is better performing hardware or a database performing better over time through improved caching.

One solution to this problem could be to clear the database of all measurements and re-execute all MQs. As the test demonstrate, this would cause a high load on the system but will reset all internal knowledge as well as bias towards any one database in Bridge-DB. Alternatively, an MQ could be executed on all databases at certain intervals of executed subsequent queries to confirm the measurements which the cost model base its decisions upon.

References

[1] Dynamic programming, February 2015. http://en.wikipedia.org/wiki/Dynamic_programming.

[2] From relational to neo4j, April 2015. http://neo4j.com/developer/graph-db-vs-rdbms/.

[3] Laravel, February 2015. http://laravel.com/.

[4] What is a graph database, April 2015. http://neo4j.com/developer/graph-database/.

[5] S. Adali, K. S. Candan, Y. Papakonstantinou, and V. Subrahmanian. Query caching and optimization in distributed mediator systems. In *ACM SIGMOD Record*, volume 25, pages 137–146. ACM, 1996.

[6] R. Angles, A. Prat-Pérez, D. Dominguez-Sal, and J.-L. Larriba-Pey. Benchmarking database systems for social network applications. In *First International Workshop on Graph Data Management Experiences and Systems*, GRADES '13. ACM, 2013.

[7] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, pages 377–387, 1970.

[8] N. Cokljat, R. S. Ettrup, C. K. Hansen, and L. Nielsen. Bridge-db: A middleware layer for distributed multi-database systems. Technical report, Department of Computer Science, Aalborg University, December 2014. Project Report.

[9] W. Du, R. Krishnamurthy, and M.-C. Shan. Query optimization in a heterogeneous dbms. In *VLDB*, volume 92, pages 277–291, 1992.

[10] T. P. G. D. Group. Web site. http://www.postgresql.org/docs/9.3/static/index.html.

[11] HAMMERDB. Web site. http://www.hammerdb.com/index.html.

[12] Intel. Intel core 2 quad processor q9450. http://ark.intel.com/products/33923/Intel-Core2-Quad-Processor-Q9450-12M-Cache-2_66-GHz-1333-MHz-FSB.

[13] H. Lu, B.-C. Ooi, and C.-H. Goh. On global multi-database query optimization. *ACM SIGMOD Record*, 21(4):6–11, 1992.

[14] C. Nance, T. Losser, R. Iype, and G. Harmon. NoSQL vs RDBMS - why there is room for both. In *Proceedings of the Southern Association for Information Systems Conference*, pages 111–116, March 2013.

[15] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems.* Springer, third edition edition, 2011.

[16] A. Rahal, Q. Zhu, and P.-Å. Larson. Evolutionary techniques for updating query cost models in a dynamic multidatabase environment. *The VLDB Journal - The International Journal on Very Large Data Bases*, 13(2):162–176, 2004.

[17] M. T. Roth, F. Ozcan, and L. M. Haas. *Cost models do matter: Providing cost information for diverse data sources in a federated system.* IBM Research Division, 1999.

[18] P. J. Sadalage and M. Fowler. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence.* Addison-Wesley Professional, 2012.

[19] Samsung. Samsung 840 evo, 2.5 sata ssd, 250 gb. http://www.samsung.com/uk/consumer/memory-cards-hdd-odd/ssd/840-evo/MZ-7TE250BW.

[20] A. Silberschatz, H. Korth, and S. Sudarshan. *Database Systems Concepts.* McGraw-Hill, Inc., 6 edition, 2011.

[21] N. Technology. Web site. http://neo4j.com/docs/2.1.5/.

[22] TPC. Tpc benchmark c. Technical report, Transaction Processing Performance Council (TPC), February 2010. http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5-11.pdf.

[23] Ubuntu. Ubuntu 15.04 (vivid vervet). http://releases.ubuntu.com/15.04/.

[24] W. Wu, Y. Chi, H. Hacígümüş, and J. F. Naughton. Towards predicting query execution time for concurrent and dynamic database workloads. *Proc. VLDB Endow.*, 6(10):925–936, Aug. 2013.

[25] Q. Zhu. Estimating local cost parameters for global query optimization in a multidatabase system. 1996.

[26] Q. Zhu and P.-A. Larson. A query sampling method for estimating local cost parameters in a multidatabase system. In *Data Engineering, 1994. Proceedings. 10th International Conference*, pages 144–153. IEEE, 1994.

[27] Q. Zhu and P.-A. Larson. Building regression cost models for multidatabase systems. In *Parallel and Distributed Information Systems, 1996., Fourth International Conference on*, pages 220–231. IEEE, 1996.

[28] Q. Zhu and P.-A. Larson. Global query processing and optimization in the cords multidatabase system. In *Proceedings of International Conference on Parallel and Distributed Computing Systems*, pages 640–646. Citeseer, 1996.

[29] Q. Zhu and P.-å. Larson. Solving local cost estimation problem for global query optimization in multidatabase systems. *Distributed and parallel databases*, 6(4):373–421, 1998.

[30] Q. Zhu, Y. Sun, and S. Motheramgari. Developing cost models with qualitative variables for dynamic multidatabase environments. In *Data Engineering, 2000. Proceedings. 16th International Conference on*, pages 413–424. IEEE, 2000.

APPENDIX

A. BRIDGE-DB PROTOCOL

The previous design used a REST API, but as this included the entire overhead of the HTTP protocol, we chose to boil the solution down to a basic TCP socket and implement our own protocol for the communication between the client and the Bridge-DB server which we named Bridge-DB Protocol (BP). Interaction with Bridge-DB is designed to expose a service on a given port using the TCP protocol. The first advantage is the limited overhead of communicating in this manner. The second advantage is, this design allows us to have bi-directional communication between the client and the Bridge-DB server. Bi-directional communication was an issue in the first version of Bridge-DB as it often was a hindrance not knowing the current state of Bridge-DB during an execution as well as debug information was delayed until the completion of the query.

Each message sent between the client and Bridge-DB contains the values seen in Table 11.

Name	Type
Sequence Number	integer
State	integer
Message	string

Table 11: Protocol values

Each message between the client and server is stamped with a *sequence number* consisting of a separator between two integers where the first integer indicates whether it is from the client or the server the messages originates, 1 and 2 respectively. The second integer indicates which step in the sequence have been reached. These sequences are used to avoid erroneous replay of messages as well as keep track of the current state of execution.

The value *State* indicates whether the execution moves forward with the value 1. In case the execution is halted, *State* will contain a negative value unique to the given step in the sequence reflecting what type of error happened.

The value *Message* can contain any additional information as well as contain the query sent from the *client* and the query result returned from *Bridge-DB.*

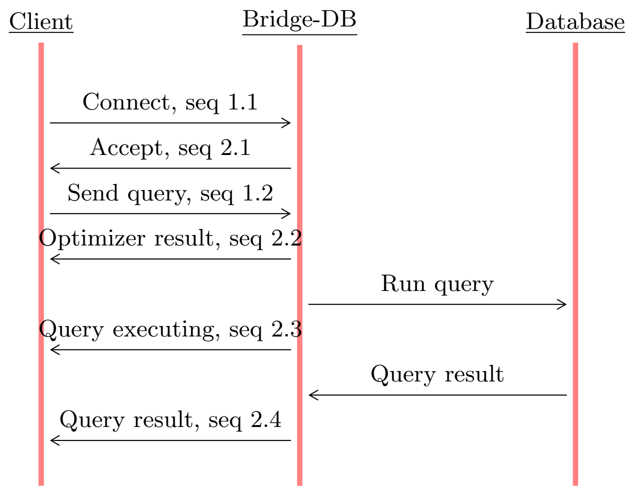
When executing a query from the *client*, the communication conforms to the sequence diagram depicted in Figure A. At each point Bridge-DB sends a message to the client, it can set the *State* value to a negative value indicating Bridge-DB assessed that the query cannot be executed, causing the execution to be halted. These assessments are based on the step Bridge-DB have been through up to the message is sent to the client.

The first rejection can happen in the *Accept* message which can be rejected if Bridge-DB is out of resources or no databases are available, each declared with a *state* indicating the issue. This allows the client application to either reconnect later or just halt all execution.

The second rejection can happen in the *Optimizer result* message indicating that the optimizer deemed the query invalid based on the internally declared database schema.

The third rejection can happen in the *Query executing* message indicating that the database is unable to process the query. This can either be a badly implemented connector or the connector rejected the query based on the database is unable to handle the query.

The fourth and last rejection can happen in the *Query result* message indicating the result is invalid. An example of an invalid result can be the database crashed during execution.



B. TRANSLATING QUERY OBJECT INTO SQL

The translation of a BQL query to SQL is simple, since BQL is highly inspired by SQL and it has a one-to-one correspondence. In this section there will be references to functions of a query objects, which can be found in Figure 26.

B.1 Create Query

An insert SQL query has typically the skeleton in Definition 7 where the name surrounded by need to inserted. The *table*,*columns* and *values* can be easily found by using the functions *getTable*,*getAllColumns* and *getAllValues* that the insert query object offers. No or little processing or reformatting of data is necessary since the queries is much alike.

DEFINITION 7. *INSERT INTO table ({columns}) VALUES ({values})*

B.2 Read

The main clauses in a SQL read query is in Definition 8 and these clauses is fully supported in the SQL translation module. The insertion of the values for *selectItems*, *fromItems* and *conditions* can be extracted from the functions *getAllSelectItems*,*getAllFromItems* and *getAllConditions*. The values for *fromItems* and *conditions* need some processing of the data since sub-queries can occur in the FROM and WHERE clause. However, the handling of the sub-queries is just like the handling of a read query, so it can be recursively translated and inserted in parenthesis in its parent query and the same is done to parenthesis sub-queries.

DEFINITION 8. *SELECT {selectItems} FROM {fromItems} WHERE {conditions}*

B.3 Update

In Definition 9 is building block of a update query. The *conditions* of the WHERE clause extracted and handled as in the read query. The data for *table* and *keyValue* can be accessed thought the functions *getTable* and *getAllUpdatePair*. No or little processing of data from the two functions are needed.

DEFINITION 9. *UPDATE {table} SET {keyValue} WHERE {conditions}*

B.4 Delete Query

Definition 10 contains an SQL delete query where the table and conditions need to be inserted. Again the conditions are handled as in the read query and the table as in the update query.

DEFINITION 10. *DELETE {table} WHERE {conditions}*

C. QUERYBUILDER

In this section explains all construct of how a BQL query is build using the *QueryBuilder*.

C.1 Create Operation

To make insertion possible in BQL two functions have been added which can be found in Table 12. *insert* needs a string as input containing the table name and it should only be called once otherwise it would just override the table name. However, to complete the insert query the *values* function should be called at least once to add data to the query, but the functions take different inputs.

Name	Input	Return Value
insert	@param: String	@return Self | Bool
values	@param: String | Array @param: Mixed | Array | QueryObject	@return Self | Bool

Table 12: Functions to make a create query

In Listing 23 are some examples of how to use the *insert* and *values* functions. If the input of *values* are strings as in line 1-2 then it is possible to call *values* again with string as input, but any other inputs would the second time be ignored and return false. In line 3-4 *values* uses array as input and in line 5-7 an array and a *QueryObject* is the input such that it is possible to support sub queries. The *QueryObject* can be generated via the function *getReadQueryAsObject* in the *QueryBuilder* and this is how subqueries is generally made in BQL.

```
1  //use values as key-value input
2  $queryBuilder->insert('user')->values('
       firstname','Mickey')->values('lastname','
       Mouse');
```

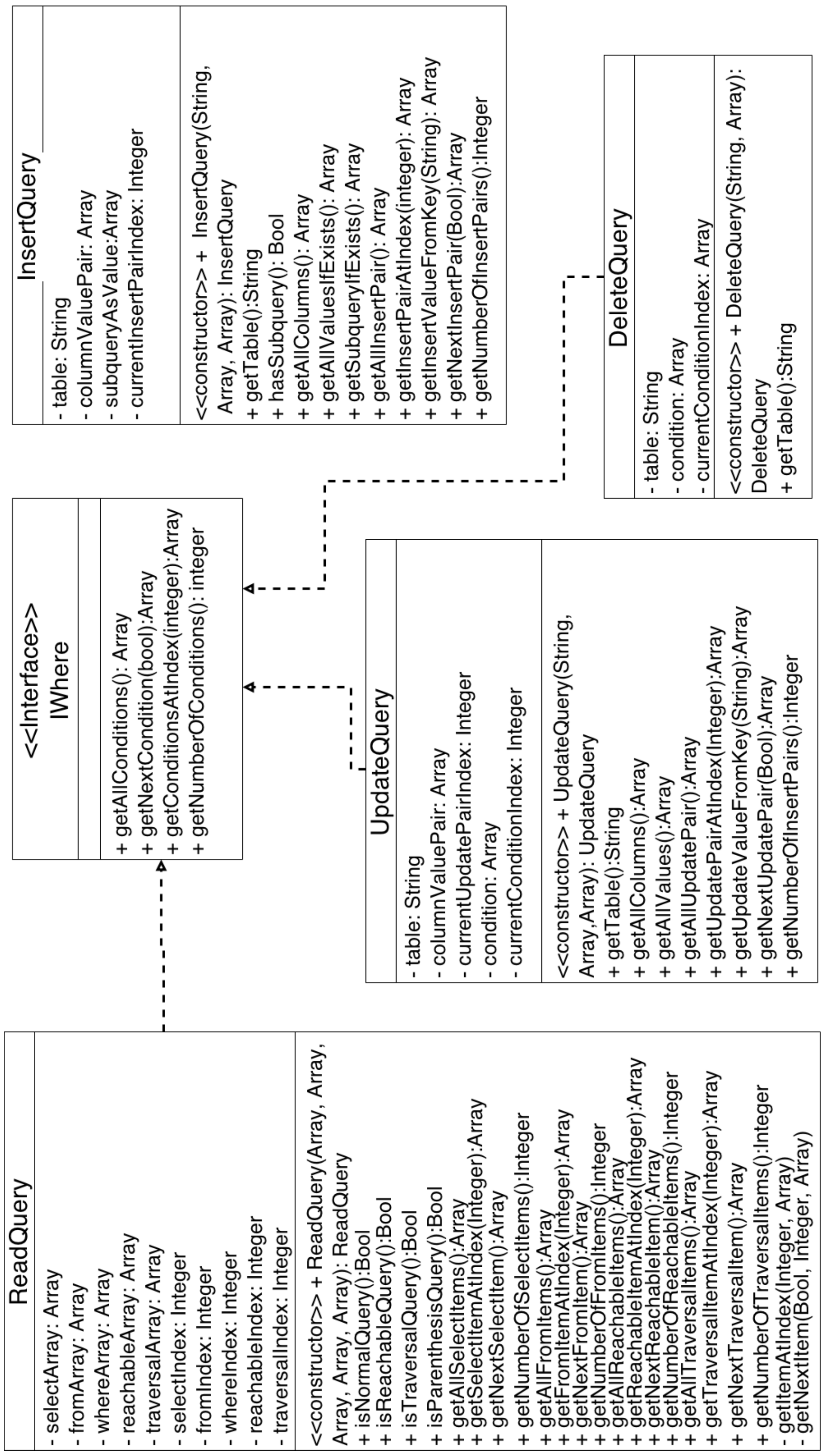

InsertQuery
- table: String
- columnValuePair: Array
- subqueryAsValue:Array
- currentInsertPairIndex: Integer
<<constructor>> + InsertQuery(String, Array, Array): InsertQuery
+ getTable():String
+ hasSubquery(): Bool
+ getAllColumns(): Array
+ getAllValuesIfExists(): Array
+ getSubqueryIfExists(): Array
+ getAllInsertPair(): Array
+ getInsertPairAtIndex(integer): Array
+ getInsertValueFromKey(String): Array
+ getNextInsertPair(Bool):Array
+ getNumberOfInsertPairs():Integer
<<Interface>>
IWhere
+ getAllConditions(): Array
+ getNextCondition(bool):Array
+ getConditionsAtIndex(integer):Array
+ getNumberOfConditions(): integer
DeleteQuery
- table: String
- condition: Array
- currentConditionIndex: Array
<<constructor>> + DeleteQuery(String, Array): DeleteQuery
+ getTable():String
UpdateQuery
- table: String
- columnValuePair: Array
- currentUpdatePairIndex: Integer
- condition: Array
- currentConditionIndex: Integer
<<constructor>> + UpdateQuery(String, Array,Array): UpdateQuery
+ getTable():String
+ getAllColumns():Array
+ getAllValues():Array
+ getAllUpdatePair():Array
+ getUpdatePairAtIndex(Integer):Array
+ getUpdateValueFromKey(String):Array
+ getNextUpdatePair(Bool):Array
+ getNumberOfInsertPairs():Integer
ReadQuery
- selectArray: Array
- fromArray: Array
- whereArray: Array
- reachableArray: Array
- traversalArray: Array
- selectIndex: Integer
- fromIndex: Integer
- whereIndex: Integer
- reachableIndex: Integer
- traversalIndex: Integer
<<constructor>> + ReadQuery(Array, Array, Array, Array, Array): ReadQuery
+ isNormalQuery():Bool
+ isReachableQuery():Bool
+ isTraversalQuery():Bool
+ isParenthesisQuery():Bool
+ getAllSelectItems():Array
+ getSelectItemAtIndex(Integer):Array
+ getNextSelectItem():Array
+ getNumberOfSelectItems():Integer
+ getAllFromItems():Array
+ getFromItemAtIndex(Integer):Array
+ getNextFromItem():Array
+ getNumberOfFromItems():Integer
+ getAllReachableItems():Array
+ getReachableItemAtIndex(Integer):Array
+ getNextReachableItem():Array
+ getNumberOfReachableItems():Integer
+ getAllTraversalItems():Array
+ getTraversalItemAtIndex(Integer):Array
+ getNextTraversalItem():Array
+ getNumberOfTraversalItems():Integer
- getItemAtIndex(Integer, Array)
- getNextItem(Bool, Integer, Array)

Figure 26: Class diagram of the InsertQuery, ReadQuery, UpdateQuery and DeleteQuery

```
//use addInsertValues as key-value input in an array
$queryBuilder->insert('user')->values(array('firstname','lastname'),array('Mickey','Mouse'));
//use values with a sub query
$queryBuilder2->select('firstname')->select('lastname')->from('olduser');
$queryBuilder->insert('user')->values(array('firstname','lastname'),$queryBuilder2->getReadQueryAsObject());
```

Listing 23: BQL Create Operation

C.2 Read

To make a read query there are many functions that can be used, see Table 13. Each of function can be called several times and it will append the additional parameters to the parameters extracted from the previous function call. Sub-queries can be included in the query either in the *from* function and different types of where-functions by using a QueryObject as input similar to the third example in Listing 23.

Name	Input	Return Value
select	@param: String	@return Self | Bool
from	@param: String | QueryObject @param: String	@return Self | Bool
where	@param: String | QueryObject @param: String @param: Mixed	@return Self | Bool
orWhere	@param: String | QueryObject @param: String @param: Mixed	@return Self | Bool
whereIn	@param: String @param: Array | QueryObject @param: Bool	@return Self | Bool
orWhereIn	@param: String @param: Array | QueryObject @param: Bool	@return Self | Bool
whereLike	@param: String @param: String @param: Bool	@return Self | Bool
orWhereLike	@param: String @param: String @param: Bool	@return Self | Bool
whereExist	@param: QueryObject @param: Bool	@return Self | Bool
orWhereExist	@param: QueryObject @param: Bool	@return Self | Bool
reachability	@param: QueryObject @param: QueryObject	@return Self | Bool
traversal	@param: QueryObject @param: QueryObject	@return Self | Bool

Table 13: Functions to make a read query

There are different types of where-functions that are able to create a condition for the query. First it can create the general condition which include a name, comparison operator and a value, see line 2 in Listing 24. Secondly the condition can also include a sub query instead of the value see line 4-5. In line 6-12 the functions creates a condition containing one of the logical operators *IN*, *LIKE*, and *EXISTS* or one of there negations which is indicated by the boolean input parameter. Lastly a condition requiring parentheses as the SQL query in Listing 25 can also be expressed with a sub query as can be found in Listing 24 in line 13-15.

```
1  //general condition
2  $queryBuilder->select('*')->from('person')->where('firstname','=','Mickey');
3
4  // condition with sub query
5  $queryBuilder2->select('username')->from('user')->where('id','=',3);
6  $queryBuilder->select('*')->from('person')->where('firstname','=',$queryBuilder2->getReadQueryAsObject());
7
8  //condition with IN
9  $queryBuilder->select('*')->from('person')->whereIn('firstname', ['Mickey','Donald']);
10 //condition with LIKE
11 $queryBuilder->select('*')->from('person')->whereLike('firstname', 'Mick%');
12
13 //condition with EXISTS
14 $queryBuilder2->select('name')->from('user')->where('user.username', '=', 'person.firstname');
15 $queryBuilder->select('*')->from('person')->whereExist($queryBuilder2->getReadQueryAsObject(), true);
16
17 //condition with parentheses
18 $queryBuilder2->where('firstname','=','Mickey')->orWhere('firstname','=','Minnie);
19 $queryBuilder->select('*')->from('person')->where('lastname','=','Mouse')->where($queryBuilder2->getReadQueryAsObject());
```

Listing 24: BQL Read Operation

```
SELECT *
FROM person
WHERE lastname = 'Mouse' AND (firstname='Mickey' OR firstname='Minnie')
```

Listing 25: The parentheses sub query

The difference between the functions *where* and *orWhere* is just the logical operator used, when there are more than one call to the function. The second time a *where* function is used and the function called is *where* then the binary operator is AND. If the second call is to *orWhere* then it uses the operator OR and this also applies to the other types of where-functions.

C.3 Update

The update query is similar to the insert query, but simpler, because the *values* function only accepts a string as input, see table 14. To make an update query the functions *update*, *values* need to be used, whereas the where-functions from Table 13 can be used. In Listing 26 is an example of updating an element in the database where the name is changed. As can be seen in the example *values* can be called multiple time to add to the query, but *update* should only be called once or it would override the table to be updated.

Name	Input	Return Value
update	@param: String	@return Self | Bool
values	@param: String @param: Mixed	@return Self | Bool

Table 14: Functions to make a update query

```
$queryBuilder->update('person')->values('
    firstname','Mickey')->values('lastname','
    Mouse')->where('firstname','=','Michey');
```

Listing 26: BQL Update Operation

C.4 Delete

The delete query is also very simple. It has a function *delete* and then it also used the different types of where-functions as in Table 13. The delete function can be found in Table 15 and an example of deleting an element from the database can be found in Listing 27.

Name	Input	Return Value
delete	@param: String	@return Self | Bool

Table 15: Functions to make a delete query

```
$queryBuilder->delete('person')->where('
    firstname', '=', 'Mickey')->where('
    lastname','=','Mouse');
```

Listing 27: BQL Delete Operation

D. MAKING THE MATCH CLAUSE

Making the MATCH clause is done in two stages, first the patterns from *where* is made into MATCH patterns. The second stage is to get the aliases of all tables that are not represented in a pattern and then add them to the MATCH clause.

Generally a pattern in MATCH can include several nodes and edge in a chain. However, when translating a pattern acquired from the *where* clause to a MATCH pattern, it will only contain three graph elements one edge and its two connecting nodes. So instead of a chained pattern it will result in subset of this chain, where a node can be present in more than one pattern, such that the chain is indirectly represented. In Listing 28 is an example of a query in BQL and how the MATCH clause can be translated into a chain or an indirectly representation of the chain. Since the returned element is not a path but specific graph elements then the pattern matching technique does not make any difference on the result, but it might in execution time depending on the optimizer in Neo4J.

```
queryBuilder->select('*')->from('user','u')->
    from('person','p1')->from('friends','f')
    ->from('person','p2')->where('p1.id','=',
    'u.person_id1')->where('p1.id','=','f.
    person_id1')->where('p2.id','=','f.
    person_id2')->where('u.username','=','
    Mickey');

MATCH (u:user)-[r1:person_id]->(p1:person)-[f
    :friends]->(p2:person) RETURN u,p1,g,p2
MATCH  (u:user)-[r1:person_id]->(p1:person),(
    p1:person)-[f:friends]->(p2:person)
    RETURN u,p1,g,p2
```

Listing 28: Equivalent match pattern

The patterns acquired from the *where* clause most be processed before it can be made into a MATCH pattern. Figure 27 shows how a pattern is asserted. A BQL example on a pattern is in the *where* clause in Listing 28. If an edge's alias is in the pattern then only the half of the MATCH pattern is represented. Two lists are needed to keep the complete patterns and the half finished patterns separated, which is called *matchList* and *halfList* in the Figure. If *halfList* contains the other half of a pattern then the two patterns can be combined and added to the *matchList*. If not then the pattern is added to *halfList*. However, if the pattern does not contain an edge then the complete pattern can be made and added to the *matchList*.

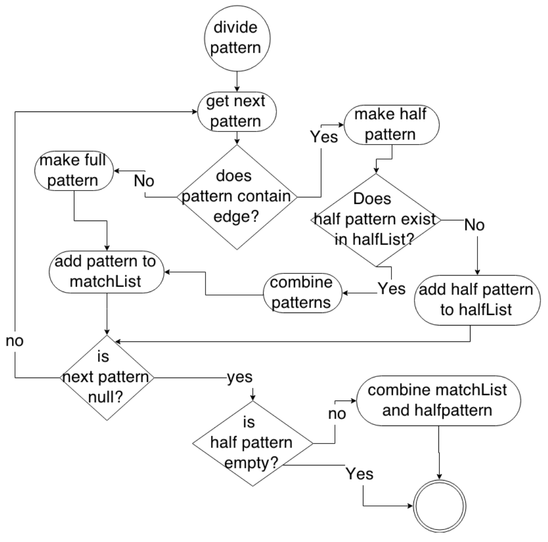


Figure 27: Make the patterns of the *where* clause into MATCH patterns

After all patterns have processed it is possible that there are still patterns in *halfList* which mean that one of the nodes in an edge is unknown which is represented with () and then the pattern can be added to the *matchList*. Finally each element in *matchList* can be added to the MATCH clause.

E. PLAN ENUMERATOR IMPLEMENTATION

In this section is the implementation details of the plan enumerator.

```
private function _split_read(\QueryObjects\
    ReadQuery $query) {
 $retr_query = array();

 $tmp_q = new \QueryBuilder\QueryBuilder();
 $fromItems = $query->getAllFromItems();
 $whereItems = $query->getAllConditions();
 $selectItems = $query->getAllSelectItems();

 foreach($selectItems as $selectItem) {
  $tmp_q->select($selectItem);
 }

 if(count($fromItems) > 1) {
  // Join operation
  for($x = 0; $x < count($fromItems); $x++) {
   $tmpObj = clone $tmp_q;
   $tmpObj->from($fromItems[$x][0]);
   foreach($whereItems as $whereItem) {
    if(preg_match("/\./", $whereItem['subject
        ']) && preg_match("/\./", $whereItem[
        'object'])) {
    } else {
     $tmpObj->where($whereItem['subject'],
         $whereItem['operator'], $whereItem['
         object']);
    }
   }
   $queries[] = $tmpObj;
  }
 } elseif(count($whereItems) > 1 && count(
     $fromItems) == 1) {
  // No join operation
  foreach($whereItems as $whereItem) {
   if(preg_match("/\./", $whereItem['subject'
       ]) && preg_match("/\./", $whereItem['
       object'])) {
   } else {
    $obj = clone $tmp_q;
    $from = $fromItems['0']['0'];
    $queries[] = $obj->from($from)->where(
        $whereItem['subject'], $whereItem['
        operator'], $whereItem['object']);
   }
  }
 } elseif(count($whereItems) == 1 && count(
     $fromItems) == 1) {
  // Simple query
  $obj = clone $tmp_q;
  $from = $fromItems['0']['0'];
  $queries[] = $obj->from($from);
 }

 if(count($whereItems) == 0 && count(
     $fromItems) == 1) {
  return $query;
 } else {
  $retr_queries[] = $query;
  foreach($queries as $query) {
   $jsonQuery = new \QueryObjects\query(
       $query->getJsonQuery());
   $retr_queries[] = $this->_split_read(
       $jsonQuery->getQuery());
  }
  return $retr_queries;
 }
}
```

Listing 29: Query split implementation

F. TEST QUERIES

This section presents all test queries as it was written via the *QueryBuilder* in BQL.

Graph Dataset

In the following listings are each query for the graph dataset presented as it was written via the *Querybuilder*.

```
$qb->select("*")->from("people")
->where('name','=','AISHA NEUENDORF')
->send();
```

Listing 30: Query 1: Find a person with a given name

```
$qb->select('people.*')
->from('people')
->from('likes')
->from('webpages')
->where('likes.id_webpage','=','webpages.
    wp_id')
->where('likes.id_person','=','people.p_id')
->where('webpages.wp_id','=',130018)
->send();
```

Listing 31: Query 2: Find all people who likes a given webpage

```
$qb->select('webpages.*')
->from('people')
->from('likes')
->from('webpages')
->where('likes.id_webpage','=','webpages.
    wp_id')
->where('likes.id_person','=','people.p_id')
->where('people.p_id','=',18)
->send();
```

Listing 32: Query 3: Find all webpages a person likes

```
$qb->select('people.name')->from('people')
->where('people.p_id','=',18)
->send();
```

Listing 33: Query 4: Find a person name from person id

```
$qb2->select('*')->from('people','p1')->where
    ('p1.p_id','=',18);
$qb3->select('*')->from('people','p2')->where
    ('p2.p_id','=',19);
$qb1->reachable($qb2->getReadQueryAsObject(),
     $qb3->getReadQueryAsObject())
->send();
```

Listing 34: Query 5: Check if there is a path between person 1 and person 2

```
$qb2->select('*')->from('people','p1')->where
    ('p1.p_id','=',18);
$qb3->select('*')->from('people','p2')->where
    ('p2.p_id','=',19);
$qb1->traverse($qb2->getReadQueryAsObject(),
    $qb3->getReadQueryAsObject())
->send();
```

Listing 35: Query 6: Find a path between person 1 and person 2

```
$qb->select(’p3.*’)
->from(’people’,’p1’)
->from(’friends’,’f1’)
->from(’people’,’p2’)
->from(’friends’,’f2’)
->from(’people’,’p3’)
->where(’p1.p_id’,’=’,’f1.id_person1’)
->where(’p2.p_id’,’=’,’f2.id_person1’)
->where(’f1.id_person2’,’=’,’p3.p_id’)
->where(’f2.id_person2’,’=’,’p3.p_id’)
->where(’p1.p_id’,’=’, 18)
->where(’p3.p_id’,’=’, 19)
->send();
```

Listing 36: Query 7: Find a common friend between two people

```
$qb->select(’w.*’)
->from(’people’,’p1’)
->from(’likes’,’l1’)
->from(’people’,’p2’)
->from(’likes’,’l2’)
->from(’webpages’,’w’)
->where(’p1.p_id’,’=’,’l1.id_person’)
->where(’p2.p_id’,’=’,’l2.id_person’)
->where(’l1.id_webpage’,’=’,’w.wp_id’)
->where(’l2.id_webpage’,’=’,’w.wp_id’)
->where(’p1.p_id’,’=’, 18)
->where(’p2.p_id’,’=’, 19)
->send();
```

Listing 37: Query 8: Find all webpages two people both like

```
$qb->select(’p3.*’)
->from(’people’,’p1’)
->from(’friends’,’f1’)
->from(’people’,’p2’)
->from(’friends’,’f2’)
->from(’people’,’p3’)
->where(’p1.p_id’,’=’,’f1.id_person1’)
->where(’p2.p_id’,’=’,’f1.id_person2’)
->where(’f2.id_person1’,’=’,’p2.p_id’)
->where(’f2.id_person2’,’=’,’p3.p_id’)
->where(’p1.p_id’,’=’, 18)
->send();
```

Listing 38: Query 9: Find all friends of friends of a person

```
$qb->select(’w.*’)
->from(’people’,’p1’)
->from(’friends’,’f1’)
->from(’people’,’p2’)
->from(’friends’,’f2’)
->from(’likes’,’l’)
->from(’webpages’,’w’)
->where(’p1.p_id’,’=’,’f1.id_person1’)
->where(’p2.p_id’,’=’,’f1.id_person2’)
->where(’p2.p_id’,’=’,’f2.id_person1’)
->where(’p2.p_id’,’=’,’f2.id_person2’)
->where(’p2.p_id’,’=’,’l.id_person’)
->where(’l.id_webpage’,’=’,’w.wp_id’)
->where(’p1.p_id’,’=’, 18)
->send();
```

Listing 39: Query 10: Find all webpages liked by friends of friends

```
$qb->select(’p2.*’)
->from(’people’,’p1’)
->from(’likes’,’l1’)
->from(’webpages’,’w’)
->from(’likes’,’l2’)
->from(’people’,’p2’)
->where(’p1.p_id’,’=’,’l1.id_person’)
->where(’l1.id_webpage’,’=’,’w.wp_id’)
->where(’l2.id_webpage’,’=’,’w.wp_id’)
->where(’p2.p_id’,’=’,’l2.id_person’)
->where(’p1.p_id’,’=’, 18)
->send();
```

Listing 40: Query 11: Find all people who likes a webpage which person with id 18 likes

Relational Dataset

In the following listings are each query for the relational dataset presented as it was written via the *Querybuilder*.

```
$qb->select(’*’)
->from(’customer’)
->where(’id’,’=’,’90032’)
->send();
```

Listing 41: Query 1: Find a customer with id 90032

```
$qb->select(’customer.*’)
->from(’district’)
->from(’customer’)
->where(’district.id’,’=’,’customer.
    district_id’)
->where(’district.id’,’=’,’3’)
->send();
```

Listing 42: Query 2: Find all customers in district 3

```
$qb->select(’*’)
->from(’customer’)
->from(’orders’)
->from(’order_line’)
->from(’stock’)
->from(’item’)
->where(’customer.id’,’=’,’90032’)
->where(’customer.id’,’=’,’orders.customer_id
    ’)
->where(’orders.id’,’=’,’order_line.order_id’
    )
->where(’order_line.stock_id’,’=’,’stock.id’)
->where(’stock.s_i_id’,’=’,’item.i_id’)
                                  ->send();
```

Listing 43: Query 3: Find all items a customer have bought

```
$qb->select(’i1.*’)
->from(’customer’,’c1’)
->from(’customer’,’c2’)
->from(’orders’,’o1’)
->from(’orders’,’o2’)
->from(’order_line’,’ol1’)
->from(’order_line’,’ol2’)
```

```
->from(’stock’,’s1’)
->from(’stock’,’s2’)
->from(’item’,’i1’)
->from(’item’,’i2’)
->where(’c1.id’,’=’,’90032’)
->where(’c2.id’,’=’,’90073’)
->where(’c1.id’,’=’,’o1.customer_id’)
->where(’o1.id’,’=’,’ol1.order_id’)
->where(’ol1.stock_id’,’=’,’s1.id’)
->where(’s1.s_i_id’,’=’,’i1.i_id’)
->where(’c2.id’,’=’,’o2.customer_id’)
->where(’o2.id’,’=’,’ol2.order_id’)
->where(’ol2.stock_id’,’=’,’s2.id’)
->where(’s1.s_i_id’,’=’,’i2.i_id’)
->where(’i1.i_id’,’=’,’i2.i_id’)
->send();
```

Listing 44: Query 4: Find all overlapping items two customers have bought

```
$qb->select(’customer.*’)
->from(’warehouse’)
->from(’stock’)
->from(’order_line’)
->from(’orders’)
->from(’customer’)
->where(’warehouse.w_id’,’=’,’3’)
->where(’warehouse.w_id’,’=’,’stock.s_w_id’)
->where(’stock.id’,’=’,’order_line.stock_id’)
->where(’order_line.order_id’,’=’,’orders.id’
    )
->where(’orders.customer_id’, ’=’, ’customer.
    id’)
->send();
```

Listing 45: Query 5: List all customers who have bought something in a warehouse 3

```
$qb->select(’item.*’)
->from(’warehouse’)
->from(’stock’)
->from(’item’)
->where(’warehouse.w_id’,’=’,’3’)
->where(’warehouse.w_id’,’=’,’stock.s_w_id’)
->where(’stock.s_i_id’,’=’,’item.i_id’)
->send();
```

Listing 46: Query 6: List all items available at warehouse 3

Merged Dataset

In the following listings are each query for the merged dataset presented as it was written via the *Querybuilder*.

```
$qb->select(’i1.*’)
->from(’customer’,’c1’)
->from(’friends’,’f1’)
->from(’friends’,’f2’)
->from(’customer’,’c2’)
->from(’orders’,’o1’)
->from(’orders’,’o2’)
->from(’order_line’,’ol1’)
->from(’order_line’,’ol2’)
->from(’stock’,’s1’)
->from(’stock’,’s2’)
->from(’item’,’i1’)
->from(’item’,’i2’)
->where(’c1.id’,’=’,’90032’)
->where(’c1.id’,’=’,’f1.id_customer1’)
->where(’f1.id_customer2’,’=’,’f2.
    id_customer1’)
->where(’f2.id_customer2’,’=’,’c2.id’)
->where(’c1.id’,’=’,’o1.customer_id’)
->where(’o1.id’,’=’,’ol1.order_id’)
->where(’ol1.stock_id’,’=’,’s1.id’)
->where(’s1.s_i_id’,’=’,’i1.i_id’)
->where(’c2.id’,’=’,’o2.customer_id’)
->where(’o2.id’,’=’,’ol2.order_id’)
->where(’ol2.stock_id’,’=’,’s2.id’)
->where(’s1.s_i_id’,’=’,’i2.i_id’)
->where(’i1.i_id’,’=’,’i2.i_id’)
->where(’i1.i_id’,’=’,’4239’)
->send();
```

Listing 47: Query 1: Find all friends of friends who have bought the same product

```
 $qb->select(’c2.*’)
->from(’customer’,’c1’)
->from(’customer’,’c2’)
->from(’friends’)
->where(’c1.id’,’=’,’90032’)
->where(’c1.district_id’,’=’,’c2.district_id’
    )
->where(’c1.id’,’=’,’friends.id_customer1’)
->where(’friends.id_customer2’,’=’,’c2.id’)
->send();
```

Listing 48: Query 2:Find all friends of a customer in the same district

```
$qb->select(’c2.*’)
->from(’customer’,’c1’)
->from(’customer’,’c2’)
->from(’friends’)
->from(’likes’)
->from(’item’)
->where(’c1.id’,’=’,’90032’)
->where(’item.i_id’,’=’,’5163’)
->where(’c1.id’,’=’,’friends.id_customer1’)
->where(’friends.id_customer2’,’=’,’c2.id’)
->where(’c2.id’,’=’,’likes.id_customer’)
->where(’likes.id_item’,’=’,’item.i_id’)
->send();
```

Listing 49: Query 3: Find all friends who have liked a given item

```
$qb->select(’c2.*’)
->from(’customer’,’c1’)
->from(’customer’,’c2’)
->from(’friends’)
->from(’likes’)
->from(’orders’)
->from(’order_line’)
->from(’stock’)
->from(’item’,’i1’)
->from(’item’,’i2’)
->where(’c1.id’,’=’,’90032’)
->where(’i1.i_id’,’=’,’5163’)
->where(’c1.id’,’=’,’friends.id_customer1’)
->where(’friends.id_customer2’,’=’,’c2.id’)
->where(’c2.id’,’=’,’likes.id_customer’)
->where(’likes.id_item’,’=’,’i1.i_id’)
->where(’c2.id’,’=’,’orders.customer_id’)
->where(’orders.id’,’=’,’order_line.order_id’
    )
->where(’order_line.stock_id’,’=’,’stock.id’)
->where(’stock.s_i_id’,’=’,’i2.i_id’)
```

```
->where('i1.i_id','=','i2.i_id')
->send();
```

Listing 50: Query 4: Find all friends who have both liked and bought an item

```
$qb->select('i1.*')
->from('customer')
->from('likes')
->from('orders')
->from('order_line')
->from('stock')
->from('item','i1')
->from('item','i2')
->where('customer.id','=','90032')
->where('i1.i_id','=','5163')
->where('customer.id','=','likes.id_customer'
    )
->where('likes.id_item','=','i1.i_id')
->where('customer.id','=','orders.customer_id
    ')
->where('orders.id','=','order_line.order_id'
    )
->where('order_line.stock_id','=','stock.id')
->where('stock.s_i_id','=','i2.i_id')
->where('i1.i_id','=','i2.i_id')
->send();
```

Listing 51: Query 5: Find all items a customer have both liked and bought

G. RAW TEST RESULTS

The raw data which the responds time diagrams are based upon, can be found in Tables 16, 17 and 18. The raw data for the data traffic diagrams can be found in Tables 19, 20 and 21.

H. QUERY EXECUTION PLANS FOR DECOMPOSED QUERIES

In Figure 28 and 29 are the QEP for query 9 and 10 in the graph dataset. Figure 30, 31 and 32 show the QEP for Query 4 and 5 in the merged dataset. As can be seen all illustrate the same pattern as described in the test results.

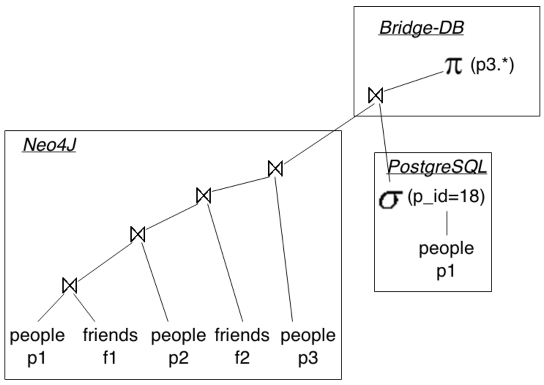


Figure 28: High-level description of QEP of query 9

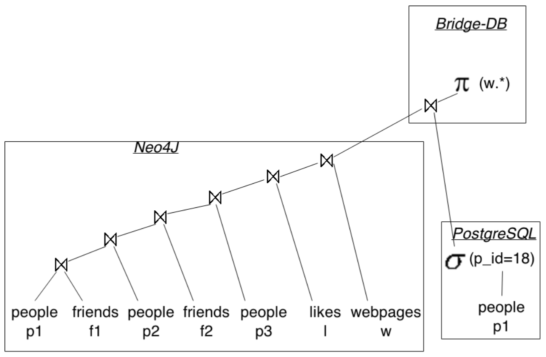


Figure 29: High-level description of QEP of query 10

I. ACRONYMS

MQ	Measurement Query
BQL	Bridge-DB Query Language
QEP	Query Execution Plan
EBNF	Extended Backus-Naur Form
DBMS	Database Management System
MDBMS	Multi Database Management System
NoSQL	Not only SQL
CRUD	Create, Read, Update and Delete
BSL	Bridge-DB Schema Language
ORM	Object/Relational Mapping
ACID	Atomicity, Consistency, Isolation and Durability
BASE	Basically Available, Soft state, Eventually consistent
CV	Credibility Value

Query	Neo4J cached	Neo4J non-cached	PostgreSQL cached	PostgreSQL non-cached	Optimizer single MQ	Optimizer multiple MQ
1	189	2899	41	86	48	45
2	11	1707	283	282	244	12
3	17	1694	241	319	331	16
4	12	1437	12	28	21	14
5	11	1945	-	-	21	12
6	12	1570	-	-	20	14
7	31	1903	446	588	71	34
8	38	1769	443	592	455	36
9	223	3145	541	903	765	232
10	32	2142	503	702	268	36
11	214	2419	1014	1324	1039	221

Table 16: Responds time in ms for the social graph dataset

Query	Neo4J cached	Neo4J non-cached	PostgreSQL cached	PostgreSQL non-cached	Optimizer single MQ	Optimizer multiple MQ
1	11	1403	12	32	19	16
2	6995	12408	8392	8387	8402	7026
3	51	1762	506	823	531	54
4	14	1633	1134	1831	436	18
5	18	1632	22597	22657	22651	22
6	1289	1567	1318	1682	1364	1342

Table 17: Responds time in ms for the relational dataset

Query	Neo4J cached	Neo4J non-cached	PostgreSQL cached	PostgreSQL non-cached	Optimizer single MQ	Optimizer multiple MQ
1	19	1654	247	324	263	24
2	29	1557	213	287	244	36
3	26	1685	212	290	144	34
4	15	1908	701	1150	346	25
5	32	1668	474	681	289	41

Table 18: Responds time in ms for the merged dataset

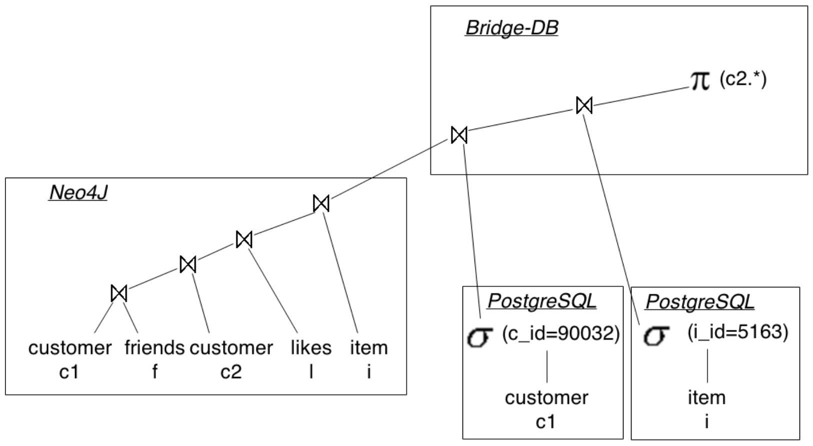


Figure 30: High-level description of QEP of query 3

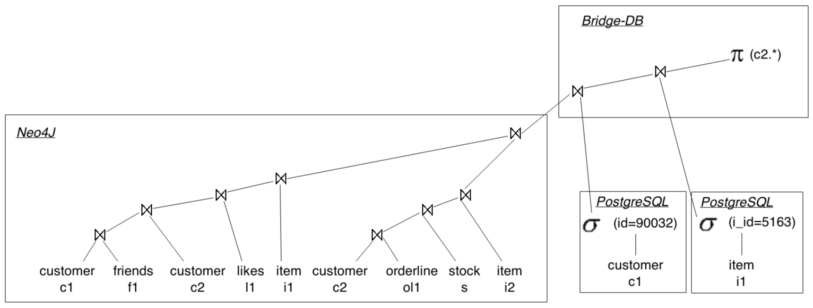


Figure 31: High-level description of QEP of query 4

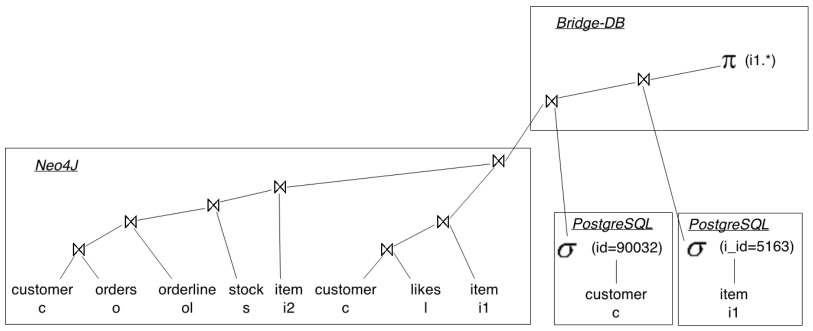


Figure 32: High-level description of QEP of query 5

Query	**From Bridge-DB**	**From Neo4J**	**From PostgreSQL**
1	1.2		
2	542.2		
3	831.5		
4	0.9		
5	0.1		
6	506		
7	787.1	4429	1.4
8	271.14		
9	441.9	14253	1.2
10	926.2	29624	1.3
11	624.1		

Table 19: Data traffic in kB for the social graph dataset

Query	**From Bridge-DB**	**From Neo4J**	**From PostgreSQL**
1	1354	20530	2.4
2	1896.7		
3	1896.7	11922	2.1
4	836.2	27638	2.3
5	2348.3	23654.8	2.2

Table 21: Data traffic in kB for the merged dataset

Query	**From Bridge-DB**	**From Neo4J**	**From PostgreSQL**
1	1.3		
2	496127		
3	1897		
4	9	340148	3.2
5	1355		
6	11997		

Table 20: Data traffic in kB for the relational dataset