UNIVERSITY OF CALIFORNIA,
IRVINE

QA Testing for a Big Data Management System: A Case Study

THESIS

submitted in partial satisfaction of the requirements
for the degree of

MASTER OF SCIENCE

in Computer Science

by

Khurram Faraaz Mohammed Noorullah Hussain

Thesis Committee:
Professor Michael J. Carey, Chair
Professor Chen Li
Assistant Professor James A. Jones

2013

© 2013 Khurram Faraaz Mohammed Noorullah Hussain

DEDICATION

To

my parents and to my wife

and to professor Michael J. Carey

AsterixDB Less in the trunk and more in the engine!

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to express the deepest appreciation to my advisor, Professor Michael Carey, he is an expert in the field of database management systems and at the same time very simple and kind hearted: he continuously provided feedback and comments on all chapters of this MS Thesis and corrected my understanding of certain core database functionality and implementations, which helped me improve the quality of the content. Every single interaction with him was a learning for me. Without his guidance and persistent help this dissertation would not have been possible. I also thank my advisor for giving me the opportunity to work with him and the Asterix team for close to two years, as a research assistant.

I would like to thank my committee members, Professor Chen Li and Assistant Professor James Jones, who took time out of their busy schedule to review my thesis and gave their valuable comments and feedback.

I would like to thank every member of the Asterix project, for having helped me understand the functionality of the different complex features. I would specifically like to mention Alexander Behm and Madhusudan for their help and support.

I also take this opportunity to thank my parents and my brother for having provided the much required financial help that was required to pursue my masters in Computer Science at University of California, Irvine. Last but not the least, without the support, sacrifice and cooperation of my wife this would have been highly impossible to achieve.

ASTERIX project is supported by an eBay matching grant, one Facebook Fellowship Award, the NSF Awards No. IIS-0910989, IIS-0910859, and IIS-0910820, a UC Discovery grant, three Yahoo! Key Scientific Challenge Awards, and generous industrial gifts from Google, HTC, Microsoft and Oracle Labs. I thank all our sponsors for supporting our project.

ABSTRACT OF THE THESIS

QA Testing for a Big Data Management System: A Case Study

By

Khurram Faraaz Mohammed Noorullah Hussain

Master of Science in Computer Science

University of California, Irvine, 2013

Professor Michael J. Carey, Chair

This thesis reports on the outcome of a six-month case study involving software testing of a large, open source code base for Big Data Management. The focus is two-fold, covering issues related to both testing a declarative query language and to the special challenges related to ensuring its correctness when data is partitioned and the execution of queries is parallelized. The system under study, AsterixDB, is a Big Data Management System (BDMS) that uses a shared-nothing parallel architecture. The parallel query processing abilities of the system, like different kinds of parallel joins (Hybrid Hash Join, Index Nested Loop Join and Nested Loop Join), aggregate functions, group by, order by, and limits, are extensively tested in a parallel shared-nothing environment.

Chapter 1

Introduction

This thesis reports on the outcome of a six-month case study involving software testing of a large, open source code base for Big Data Management. The focus is two-fold, covering issues related to both testing a declarative query language and to the special challenges related to ensuring its correctness when data is partitioned and the execution of queries is parallelized. The system under study, AsterixDB, is a Big Data Management System (BDMS) that uses a shared-nothing parallel architecture. The parallel query processing abilities of the system, like different kinds of parallel joins (Hybrid Hash Join, Index Nested Loop Join and Nested Loop Join), aggregate functions, group by, order by, and limits, are extensively tested in a parallel shared-nothing environment.

The remainder of this thesis is organized as follows. We discuss work related to testing parallel database systems and their querying capabilities in Chapter 2. We introduce the readers to AsterixDB and its architecture in brief in Chapter 3. In Chapter 4, we give a high-level description of the different parallel partitioned query processing capabilities of AsterixDB. In Chapter 5, we give a detailed description of the strategic test plan that we have used to execute and test the parallel partitioned query processing capabilities of AsterixDB. This test plan is a high-level description of the different test scenarios that are applicable to the parallel query processing algorithms of AsterixDB. In Chapter 6, we provide an overview of the our experiences, results and reflections on

interesting test scenarios. Finally, we conclude in Chapter 7 with a summary and possible further action that can be pursued in this direction.

Chapter 2

Related Work

Testing and quality assurance have a long-standing place in database system development [34]. The practice of testing database management systems has led to the design and development of several interesting strategies and frameworks to verify and validate the correctness of many different DBMS functionalities, as evidenced by publications in previous editions of the DBTest workshop series [46]. There have been significant contributions in the area of testing of parallel shared-nothing database management systems, both from academia and industry.

In this section we briefly introduce readers to existing research in the area of database management systems testing. Testing any database management system involves (i) functional test practices that verify and validate the correctness of the functionality of individual components, as well as (ii) system test practices that test the database system as a whole end-to-end. Some of the interesting topics in the field of testing of parallel database management systems are generating random test queries for execution on a given target database system [25], generating huge volumes of data over which many different queries can be executed [36], generating query-aware databases (e.g., QAGen [35]), and testing the accuracy of query optimizers [45]. We review each of these briefly.

Deterministic testing of SQL-based database systems is human-intensive and cannot adequately cover the entire SQL domain. As a response to the need for

automated AQL (Asterix Query Language) testing, a system called RAGS (Random Generation of SQL) was built to stochastically generate valid SQL statements and execute them a million times faster than a human could [25]. Typical SQL test libraries contain tens of thousands of statements, and it requires an estimated ½ person hour to compose a correct query in SQL [25] or any declarative query language. This is why the testing of large DBMS software takes a long time; the release of such DBMS software is dependent on when the testing is declared as being complete. This adds a significant cost to the overall development cost of the product.

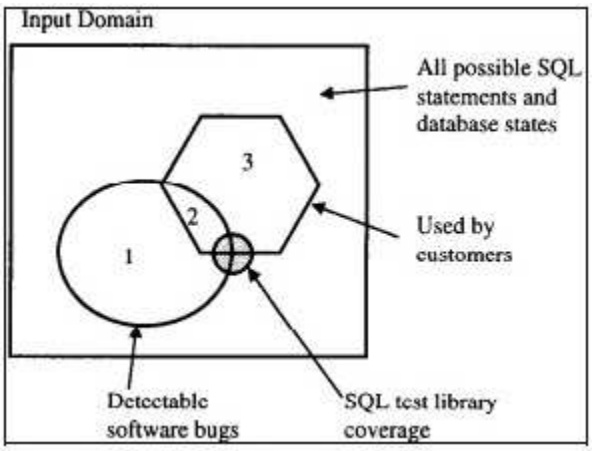

Figure 1: SQL test library coverage should cover region 2 as much as possible.

Figure 1 [25] illustrates the problem of SQL test coverage. Customers tend to use the area marked in the hexagon, while bugs are present in the oval, and test libraries cover the shaded small circle. Unfortunately, we don't know the actual region boundaries [25]. To increase the area covered by the test libraries, statements are generated stochastically (or 'randomly') which provides the speed as well as wider coverage of the input domain [25]. In a nutshell, RAGS generates SQL statements by randomly walking

a stochastic SQL parse tree and printing queries out. RAGS was able to steadily uncover bugs in released SQL products.

Generating huge volumes of related data over a cluster of nodes is another interesting topic, where an attempt is made to match the quantity and the quality of the data that is handled by production systems in large enterprises. Running realistic benchmarks to test the performance and robustness of these applications is becoming increasingly difficult because of the amount of data that needs to be generated, the number of systems that need to generate the data, and the complex structure of the data [36].

PSDG is a parallel synthetic data generator designed to generate “industrial sized” data sets quickly using cluster computing. PSDG depends on SDDL, a synthetic data description language that provides flexibility in the types of data that can be generated [48]. SDDL is an XML-based language that codifies the manner in which generated data can be described and constrained. PSDG is designed to generate data across multiple processors. Each PSDG data generation process is launched with the knowledge of how many processes are participating as well as its own process index. With this information, a generation process can determine the extent of the data that it is responsible for generating without the need for inter-process communication. PSDG slices the generated data horizontally between generation processes. Data generation processes handle slices in a “striped” fashion: process 0 of N will generate slices {0, N, 2N,…}, process 1 of N will generate slices {1, N+1, 2N+1,…}, and so on. PSDG is written in Java and it supports direct-to-database data generation; it can also generate data to file(s).

As an example, passing the SSDL snippet in Figure 2 as a file to PSDG would result in the relation named PAIRS which looks like Table 1.

```
<?xml version="1.0" encoding="UTF-8"?>
<database>
        <seed>1240958412</seed>
        <pool name=”colors”>
                <choice=”red”/>
                <choice=”green”/>
                <choice=”blue”/>
        </pool>
        <table name="PAIRS" length="5">
                <field name="F1" type="int">
                        <min>1</min>
                        <max>50</max>
                </field>
                <field name="F2" type="CHAR(5)">
                        <formula>colors</formula>
                </field>
        </table>
</database>
```

Figure 2 : SSDL snippet

Table 1: Generated relation PAIRS [48]

F1	**F2**
34	green
10	blue
47	green
17	red
8	red

QAGen [35], summarised in Figure 3, is a query aware database generator that generates test databases. It operates in two phases: (1) a symbolic query processing

phase (SQP) and (2) a data instantiation phase. QAGen applies the concept of symbolic execution from software engineering to traditional query processing. Symbolic execution is a well-known program verification technique that represents values of program variables with symbolic values instead of concrete data and then manipulates expressions based on those symbolic values [49]. Using this concept, QAGen first instantiates a database which contains a set of symbols instead of concrete data (thus the generated database in this phase is called a symbolic database). In the SQP phase, the input query is analyzed by a query analyzer. Then, users specify their desired constraints and knobs (these knobs allow a user to control the output size of an operator) on the operators of the query tree. Later the input query is executed by a symbolic query engine. In the data instantiation phase that follows the SQP phase, symbolic tuples are read from the symbolic database and used to instantiate the symbols with real data values. The instantiated tuples are then inserted into the target database [35].

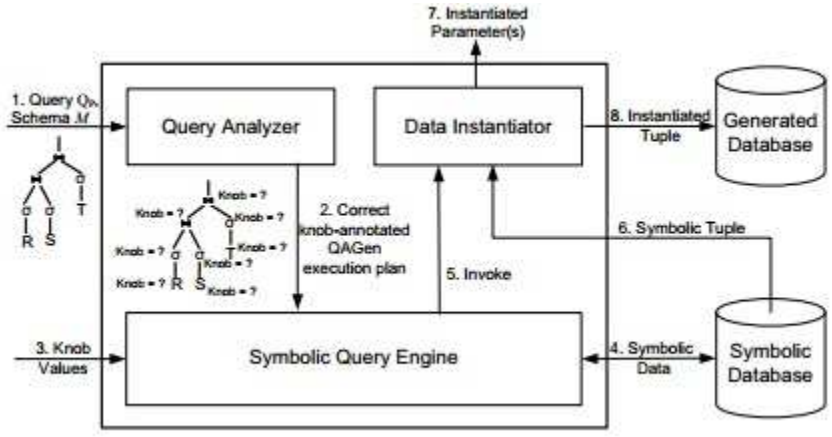

Figure 3: QAGen Architecture [35]

Testing the correctness of query optimizers is another very interesting area in the field of testing database management systems [45]. The accuracy of a query optimizer is intricately connected with a parallel database system's performance and its operational cost: the more accurate the optimizer's cost model, the better the resulting query execution plans. When an optimizer consistently ranks plan alternatives correctly, the plan distribution plot shows a monotonic trend. That is, as the estimated plan cost increases, the actual cost also increases. In practice, no optimizer achieves perfect accuracy for non-trivial queries. Hence, instead of checking whether or not an optimizer is perfectly accurate, one can develop a metric that allows accuracy to be measured in a nuanced way. In other words, given a query Q and a sample SQ = {p1,...,pn} of plans from the plan space, the goal of [45] was to compute a correlation score between the ranking of plans in SQ based on estimated costs and their ranking based on actual costs.

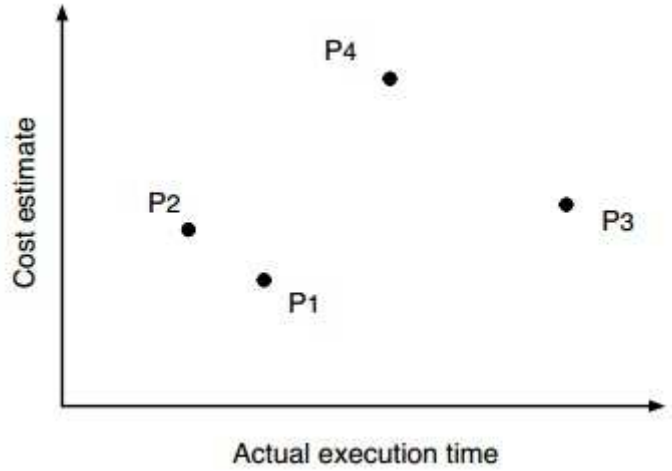

Figure 4: Scatter plot for plan alternatives [45]

As noted in [45], if the actual execution cost for two plans is very close (i.e the pairwise distance of two points is small), we might not be able to order them

conclusively in an experiment. In this case, a ranking mistake for two such plans should not be weighted the same as if they were affiliated with two distant points. For example in Figure 4, incorrectly ranking the plan pair (p1, p2) is less significant than getting pair (p3, p4) wrong [45].

In addition to these works on DBMS testing, there is also a different kind of testing, namely performance testing, that Big Data systems need and are undergoing [58, 59, 60]. That kind of testing is orthogonal to correctness testing and plan testing and is outside the scope of this thesis.

In this chapter we have discussed several different areas related to software testing of database management systems. The focus of this thesis is towards verifying the correctness of the different parallel query processing capabilities of AsterixDB in a parallel environment at a functional test level. Coverage includes scenarios that test the correctness of the way that AQL statements are handled by the parser and runtime system, and for some interesting scenarios the optimized query plans are also verified for correctness. We will see in the chapters that follow what the coverage was like and what some of the interesting findings were from this work. As part of the study the existing test framework to test AsterixDB was also updated to use the system's new HTTP-based API for DDL and DML statement execution and verification of results.

Chapter 3

Overview of AsterixDB

A wealth of digital information is being generated daily, information has great potential value for many purposes if captured and aggregated effectively. In the past, data warehouses were largely an enterprise phenomenon, with large companies being unique in recording their day-to-day operations in databases and in warehousing and analyzing historical data to improve their businesses. Today, organizations and researchers in a wide variety of areas are seeing tremendous potential value and insight to be gained by warehousing the emerging wealth of digital information, popularly referred to as “big data,” and making it available for analysis, querying, and other purposes [50].

AsterixDB has been built as a next-generation data management system in response to these trends. AsterixDB was designed by combining and extending ideas from semi-structured data management, parallel database systems, and first-generation data-intensive computing platforms such as MapReduce and Hadoop. It is a parallel, shared-nothing, semi-structured Big Data Management System (BDMS), with the ability to ingest, store, index, query and analyze very large quantities of semi-structured data [51]. AsterixDB is well-suited for use cases ranging from "data" use cases - where information is well-typed and highly regular - to "content" use cases - where data tends to be irregular, much of each datum may be textual, and the ultimate schema for the various data types involved may be hard to anticipate up front.

Figure 5 provides an overview of how the different components of AsterixDB map to nodes in a shared-nothing cluster. The bottom-most layer provides storage facilities for managed datasets based on LSM-trees, which can be targets of data loads, users' inserts or continuous data ingestion. Further up the stack is Hyracks, which is a data-parallel runtime [51]. It sits at roughly the same level as Hadoop does for high-level languages such as Pig [23], Hive or Jaql [11]. The topmost layer of AsterixDB is a parallel DBMS, with a full, flexible data model (ADM) and query language (AQL) for describing, querying and analyzing data. AQL and ADM support both the native storage and indexing of data as well as access to external data (e.g., in HDFS).

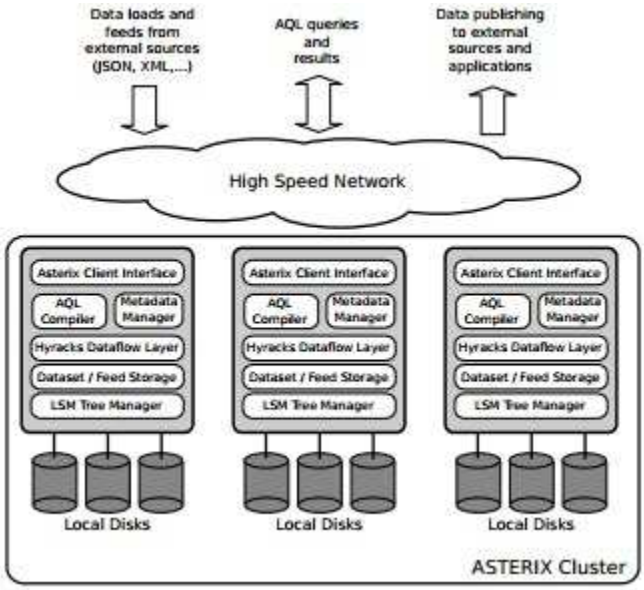

Figure 5: AsterixDB system architecture. [51]

Figure 6 provides more details about the open-source AsterixDB software stack, which supports AsterixDB but also aims to address other Big Data requirements. To process AQL queries, AsterixDB compiles each query into an Algebricks algebraic program. Algebricks is a model-agnostic, algebraic "virtual machine" for optimizing

parallel queries. The program is then optimized via algebraic rewrite rules that reorder the Algebricks operators as well as introducing partitioned parallelism for scalable query execution; code generation then translates the resulting physical query plan into a corresponding Hyracks job that uses the Hyracks runtime to compute the desired query result. The Algebricks layer is data-model-neutral and is therefore able to support other high level languages such as Hive and XQuery.

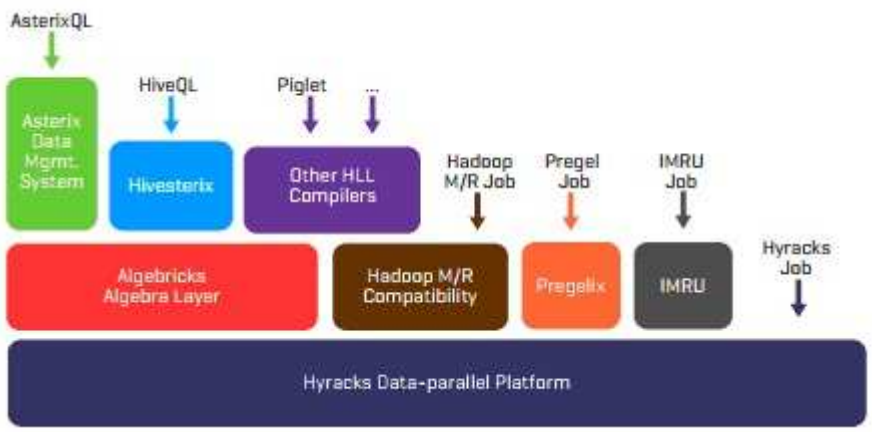

Figure 6: AsterixDB software stack [52]

The Asterix data model (ADM) borrowed data concepts from JSON [10] and added more primitive types as well as type constructors from semi-structured and object databases.

```
create type TweetType
as open {
  id: string,
  username: string,
  location: point?,
  text: string,
  timestamp: string,
  hashTags: {{string}}?
};
```

```
create type NewsType
as open {
  id: string,
  title: string,
  description: string,
  link: string,
  topics: {{string}}?
};
```

Figure 7: ADM Types to represent Tweets & News articles as records [51]

Figure 7 illustrates ADM by showing how it could be used to define a record type for modeling Twitter messages. The record type TweetType is defined as an open type, meaning that its instances should conform to its specification but will also be allowed to contain arbitrary additional fields that can vary from one instance to another. The field "location" is defined to be optional, and the field "hashTags" is defined as a nested collection of string values.

Logical data storage in AsterixDB is based on the concept of a "dataset", which is a declared collection of instances of a given type. Internal datasets are stored and managed as partitioned LSM-based B+ trees with optional secondary indexes, while external datasets that can reside in local or HDFS files. Data is hash-partitioned across different nodes based on the primary key defined in the create dataset DDL statement [51]. Given the usefulness of tracking and analyzing data from social media, AsterixDB has another powerful feature that can be used to continuously ingest huge volumes of data so that later analysis can be performed on that data. Data can be stored in feed datasets from many different sources using adapters in AsterixDB; these adapters abstract away the mechanism of connecting with an external service.

To illustrate ADM further, consider the create type and create dataset DDL statements, which create an open type named EmpType and a dataset named Employee which is defined to be of EmpType.

```
create type EmpType as open {
        id: int32,
        name: string,
        age: int8,
        dept-name: string
}
create dataset Employee(EmpType) primary key id;
```

The dataset Employee is defined to be of EmpType in the example, and it will be hash partitioned across the nodes in the cluster based on the primary key “id”. If there was a predicate of the form “where $l.id = 167”, the system would go to the exact node where the tuple with employee id = 167 was stored to fetch it. AsterixDB also allows users to define optional secondary indexes (e.g., a B+ Tree or RTree index), e.g., on the “age” field in our example. Every node in the cluster would be accessed in parallel if the predicate was a range predicate over age, such as:

```
for $l in dataset Employee
where $l.age > 18 and $l.age < 30
return $l
```

In this case, each node would use the age index to determine the primary keys of the Employee instances in the specified range and then retrieve the Employees via the corresponding local primary index partition.

As we will see in the next section, AQL is a rich query language that includes support for selections, joins, nested queries, aggregates, grouping, ordering and more, leading to a challenging correctness testing task. This task is exacerbated by the data

partitioning, primary and secondary index types, and data typing options that ADM offers.

Chapter 4

Parallel Partitioned Query Processing in AsterixDB

In this chapter we will discuss the different parallel query processing abilities of AsterixDB, some of which are joins, aggregates, grouping, ordering and limits. Each of those features is explained in the next sections, with a simple AQL example that illustrates that feature, and the query plan for that query is also discussed.

4.1 Joins

A join in a database management system is an operation where we combine two tuples from two relations R and S based on a certain join predicate. A join of the form $R \bowtie_{r.A = s.B} S$ is referred to as an equi-join, where A and B are attributes of relations R and S, respectively. Large equi-joins are commonly implemented as hash-based joins in database systems. In this chapter we will discuss parallel joins and how they work in AsterixDB. Multiprocessor hash based join algorithms are discussed in detail in [42]. In the following sections we will discuss details of parallel join methods such as Hybrid Hash Join, Indexed Nested Loops Join, and Block Nested Loops Join in AsterixDB at a high level.

4.1.1 Hybrid Hash Join

Before we get to Hybrid Hash Joins, dynamic Hybrid hash algorithms in general [13] are described in this section. Dynamic Hybrid hash algorithms start out with the (optimistic) premise that no overflow will occur; if it does, however, they partition the input into multiple partitions of which only one is written immediately to temporary files on disk. The other F-1 partitions remain in memory. If another overflow occurs, another

partition is written to disk. If necessary, all F partitions are written to disk. Thus, hybrid hash algorithms use all available memory for in-memory processing, but at the same time are able to process large input files by overflow resolution, Figure 8 shows the idea of hybrid hash algorithms. As many hash buckets as possible are kept in memory, e.g., as linked lists as indicated by solid arrows. The other hash buckets are spooled to temporary disk files, called the overflow or partition files, and are processed in later stages of the algorithm. Hybrid hashing is useful if the input size R is larger than the memory size M but smaller than the memory size multiplied by the fan-out F, i.e., $M < R < F \times M$, [13].

The parallel version of the Hybrid hash-join algorithm divides the join problem into parallel tasks using hashing and then performs the local partitioning and joining phases on the same join nodes. Each processor partitions its portion of the source relations using the same hash function. Each node allocates excess memory during the partitioning phase to a hash table for one bucket of tuples. As the source relations are being hash-partitioned, most tuples are written across the network to the appropriate join node. Tuples belonging to the bucket associated with a partitioning processor are instead immediately used to either build or probe the local hash table [42]. Figure 9 illustrates a partitioned join in a multiprocessor system.

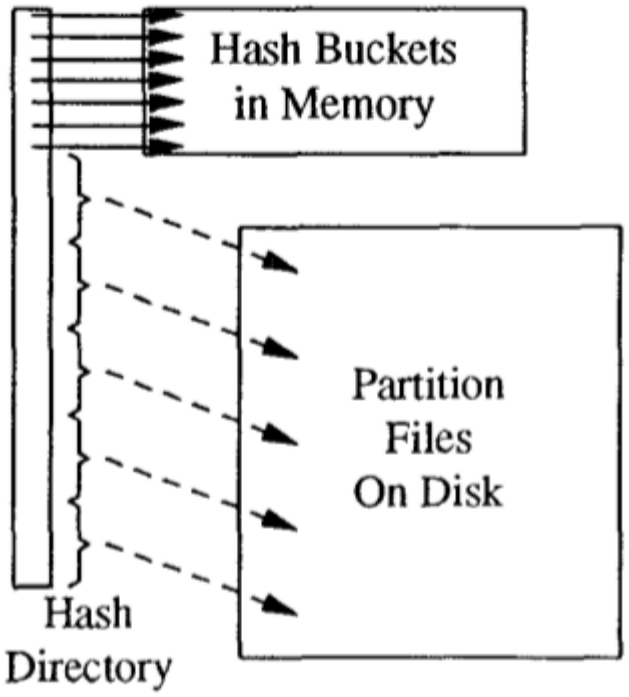

Figure 8: Hybrid Hashing [13]

If one or both of the relations R and S are already hash partitioned on the join attributes, the work needed for partitioning is reduced greatly. If the relations are not partitioned, or are partitioned on attributes other than the join attributes, then the tuples need to be repartitioned. Skew presents a special problem; however, with a good hash function, hash partitioning is likely to have a smaller skew, except when there are many tuples with the same values for the join attributes [54].

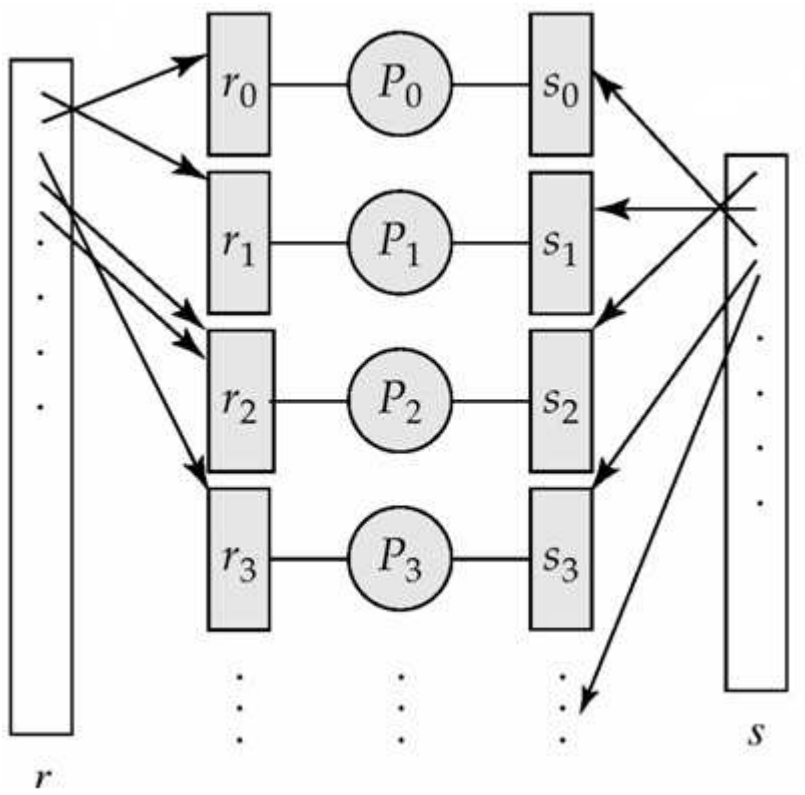

Figure 9: Partitioned Join [54]

Figure 10 illustrates an AQL query that performs an equijoin on Customers and Orders datasets. The Customers dataset has a primary key defined on its “cid” attribute and Orders dataset has a primary key defined on its “oid” attribute. The optimized query plan for the equi-join AQL query is available in Figure 11. A data source scan (i.e. scan a dataset) is performed on the Customers dataset, followed by a one to one exchange of data between source and destination nodes, then a stream project is performed to retain only the required attributes of the Customers dataset. A data source scan is performed on the Orders dataset and it is repartitioned, because Orders.cid is not the primary key in Orders dataset. A hybrid hash join is then performed at each node on its local fragment of Customers and the received fragment of the Orders dataset. A project

is later performed to project only the required tuples for the required attributes, and finally the results are returned.

```
// DDL to create an open type Cust
create type Cust
as open {
        cid: int32,
        name: string
}

// DDL to create an other open type Ord
create type Ord
as open {
        oid: int32,
        cid: int32,
        shipToName: string
}

// DDL statements to create Customers & Orders datasets.
create dataset Customers(Cust) primary key cid;
create dataset Orders(Ord) primary key oid;

// Insert data into Customers & Orders datasets.

// AQL Query - Equi-join
for $o in dataset Orders
for $c in dataset Customers
where $c.cid = $o.cid
return {"name":$c.name}
```

Figure 10: AQL Query to perform Equijoin in AsterixDB

```
distribute result [%0->$$7]
-- DISTRIBUTE_RESULT  |PARTITIONED|
  exchange
  -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
    project ([$$7])
    -- STREAM_PROJECT  |PARTITIONED|
      assign [$$7] <- [function-call: asterix:open-record-constructor, Args:[A
String: {name}, %0->$$13]]
      -- ASSIGN  |PARTITIONED|
        project ([$$13])
        -- STREAM_PROJECT  |PARTITIONED|
          exchange
          -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
            join (function-call: algebricks:eq, Args:[%0->$$10, %0->$$12])
            -- HYBRID_HASH_JOIN [$$12][$$10]  |PARTITIONED|
              exchange
              -- HASH_PARTITION_EXCHANGE [$$12]  |PARTITIONED|
                project ([$$12, $$13])
                -- STREAM_PROJECT  |PARTITIONED|
                  assign [$$13, $$12] <- [function-call: asterix:field-access-
by-name, Args:[%0->$$0, AString: {name}], function-call: asterix:field-access-
by-index, Args:[%0->$$0, AInt32: {1}]]
                  -- ASSIGN  |PARTITIONED|
                    project ([$$0])
                    -- STREAM_PROJECT  |PARTITIONED|
                      exchange
                      -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                        data-scan []<-[$$9, $$0] <- test:Orders
                        -- DATASOURCE_SCAN  |PARTITIONED|
                          exchange
                          -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                            empty-tuple-source
                            -- EMPTY_TUPLE_SOURCE  |PARTITIONED|
              exchange
              -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                project ([$$10])
                -- STREAM_PROJECT  |PARTITIONED|
                  exchange
                  -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                    data-scan []<-[$$10, $$1] <- test:Customers
                    -- DATASOURCE_SCAN  |PARTITIONED|
                      exchange
                      -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                        empty-tuple-source
```

Figure 11: Optimized Query Plan for equi-join in AsterixDB that uses Hybrid Hash Join method

4.1.2 Indexed Nested Loops Join

The goal of implementing an indexed nested loops join is to reduce the amount of I/O by replacing the inner join loop of a basic nested loops join by an index lookup. This replacement greatly reduces the number of I/O's that would have been otherwise required to scan the entire inner relation to look for matching tuples. However, this reduction is achieved at a price, as the sequential I/O spent in scanning the inner relation is now turned into random I/O due to the index lookup. The algorithm for an indexed nested loops join on a single node is depicted in Figure 12.

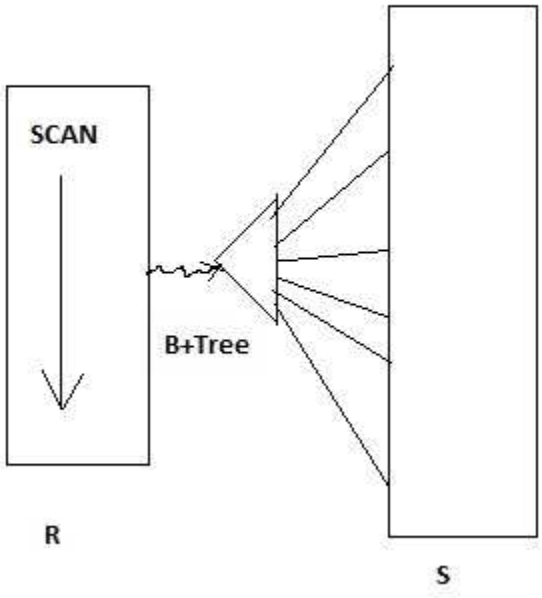

Figure 12: Indexed nested loops join.

In the case of a parallel indexed nested loops join, to exploit the degree of parallelism, both R and S are declustered across the nodes in the cluster. To effect the full join, every tuple from relation R (outer relation) is broadcast to every node in the

cluster. After R is broadcast to every node, every node in the cluster joins R with its local fragment of S, using indexed nested loops to perform the local join [56].

An important improvement available in AsterixDB when performing an indexed nested loops join of R and S against a primary index on S is that R is not broadcast. Instead, the tuples of R are hash partitioned because we know that a dataset is partitioned on its primary key (and we are probing S's primary index in the join).

```
// DDL to create an open type Cust
create type Cust
as open {
        cid: int32,
        name: string
}


// DDL to create an other open type Ord
create type Ord
as open {
        oid: int32,
        cid: int32,
        shipToName: string
}


// DDL to create Customers & Orders datasets.
create dataset Customers(Cust) primary key cid;
create dataset Orders(Ord) primary key oid;


// DDL to create secondary index on Orders.shipToName
create index idx-nm on Orders(shipToName);

// Insert data into Customers and Orders datasets.

// AQL Query - Indexed Nested Loops Join
for $o in dataset Orders
for $c in dataset Customers
where $c.name /* +indexnl */ = $o.shipToName
return {"name":$o.shipToName}
```

Figure 13: AQL to perform Indexed Nested Loops Join in AsterixDB

```
distribute result [%0->$$7]
-- DISTRIBUTE_RESULT  |PARTITIONED|
  exchange
  -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
    project ([$$7])
    -- STREAM_PROJECT  |PARTITIONED|
      assign [$$7] <- [function-call: ASTERIX_PRIVATE:closed-record-constructo
r, Args:[AString: {name}, %0->$$9]]
      -- ASSIGN  |PARTITIONED|
        project ([$$9])
        -- STREAM_PROJECT  |PARTITIONED|
          select (function-call: algebricks:eq, Args:[%0->$$12, %0->$$9])
          -- STREAM_SELECT  |PARTITIONED|
            project ([$$9, $$12])
            -- STREAM_PROJECT  |PARTITIONED|
              assign [$$9] <- [function-call: ASTERIX_PRIVATE:field-access-by-
index, Args:[%0->$$0, AInt32: {2}]]
              -- ASSIGN  |PARTITIONED|
                project ([$$0, $$12])
                -- STREAM_PROJECT  |PARTITIONED|
                  exchange
                  -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                    unnest-map [$$10, $$0] <- function-call: ASTERIX_PRIVATE:i
ndex-search, Args:[AString: {Orders}, AInt32: {0}, AString: {test}, AString: {
Orders}, ABoolean: {true}, ABoolean: {false}, AInt32: {1}, %0->$$14, AInt32: {
1}, %0->$$14, TRUE, TRUE, FALSE]
                    -- BTREE_SEARCH  |PARTITIONED|
                      exchange
                      -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                        order (ASC, %0->$$14)
                        -- STABLE_SORT [$$14(ASC)]  |PARTITIONED|
                          exchange
                          -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                            project ([$$12, $$14])
                            -- STREAM_PROJECT  |PARTITIONED|
                              exchange
                              -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                                unnest-map [$$13, $$14] <- function-call: ASTE
RIX_PRIVATE:index-search, Args:[AString: {idx-nm}, AInt32: {0}, AString: {test
}, AString: {Orders}, ABoolean: {true}, ABoolean: {true}, AInt32: {1}, %0->$$1
2, AInt32: {1}, %0->$$12, TRUE, TRUE, TRUE]
                                -- BTREE_SEARCH  |PARTITIONED|
                                  exchange
                                  -- BROADCAST_EXCHANGE  |PARTITIONED|
                                    project ([$$12])
                                    -- STREAM_PROJECT  |PARTITIONED|
                                      assign [$$12] <- [function-call: ASTERIX
_PRIVATE:field-access-by-index, Args:[%0->$$1, AInt32: {1}]]
                                      -- ASSIGN  |PARTITIONED|
                                        project ([$$1])
                                        -- STREAM_PROJECT  |PARTITIONED|
                                          exchange
                                          -- ONE_TO_ONE_EXCHANGE  |PARTITIONED
|
                                            data-scan []<-[$$11, $$1] <- test:
Customers
                                            -- DATASOURCE_SCAN  |PARTITIONED|
                                              exchange
                                              -- ONE_TO_ONE_EXCHANGE  |PARTITI
ONED|
                                                empty-tuple-source
                                                -- EMPTY_TUPLE_SOURCE  |PARTIT
```

Figure 14: Optimized Query Plan for Indexed Nested Loops Join

An indexed nested loops join in AsterixDB is illustrated in Figure 13, and the corresponding optimized query plan for that query is available in Figure 14. A data source scan is performed on the Customers dataset, and then a project provides only the required attributes to the next level of operators. The Customers dataset is then broadcast-exchanged with every other node. An index scan (B+ Tree index lookup) is performed using the secondary index "idx-nm" defined on Orders.shipToName attribute to look for a match in the Orders dataset. A sort is then performed, so that the primary index is looked up efficiently in increasing order of the primary keys. For each match found, another index scan is performed using the value received from the earlier secondary index lookup, this time on the primary index defined on Orders dataset. Then a final project is performed to project only the required attributes to the select operator and the results are returned.

4.1.3 Block Nested Loops Join

The algorithm for a block nested loops join is similar to a basic tuple oriented nested loops join algorithm, except that we compare the outer and the inner relations block by block instead of doing a tuple by tuple comparison. Figure, 15 shows this in more detail, in a parallel setup every R block is broadcast to every node in the cluster. After broadcasting R, every node in the cluster joins R with its local fragment of S, using block nested loops to perform the local join [56].

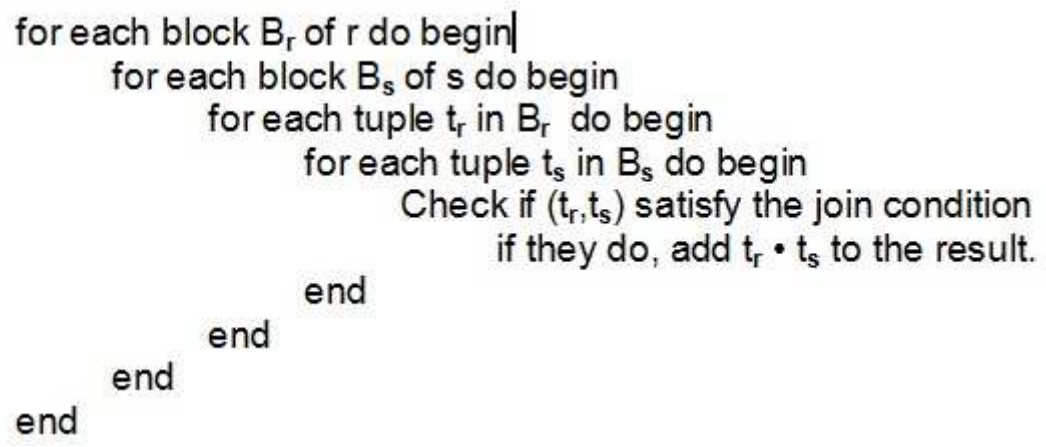

Figure 15: Algorithm - Block Nested Loops Join [54]

Block nested loops joins are used for non-equi-joins in AsterixDB. Figure, 16 shows an AQL query that performs a block nested loops join on the Orders and Customers datasets and Figure 17 shows the query plan.

```
// DDL to create an open type Cust
create type Cust
as open {
        cid: int32,
        name: string
}

// DDL to create an other open type Ord
create type Ord
as open {
        oid: int32,
        cid: int32,
        shipToName: string
}
// Create Customers and Orders datasets
create dataset Customers(Cust) primary key cid;
create dataset Orders(Ord) primary key oid;

// AQL - Block Nested Loops Join
for $o in dataset Orders
for $c in dataset Customers
where $c.cid > $o.cid
return {"name": $c.name}
```

Figure 16: AQL query that performs Block Nested Loops Join in AsterixDB

```
distribute result [%0->$$7]
-- DISTRIBUTE_RESULT  |PARTITIONED|
  exchange
  -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
    project ([$$7])
    -- STREAM_PROJECT  |PARTITIONED|
      assign [$$7] <- [function-call: asterix:closed-record-constructor, Args:
[AString: {name}, %0->$$13]]
      -- ASSIGN  |PARTITIONED|
        project ([$$13])
        -- STREAM_PROJECT  |PARTITIONED|
          exchange
          -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
            join (function-call: algebricks:gt, Args:[%0->$$10, %0->$$12])
            -- NESTED_LOOP  |PARTITIONED|
              exchange
              -- BROADCAST_EXCHANGE  |PARTITIONED|
                project ([$$12])
                -- STREAM_PROJECT  |PARTITIONED|
                  assign [$$12] <- [function-call: asterix:field-access-by-ind
ex, Args:[%0->$$0, AInt32: {1}]]
                  -- ASSIGN  |PARTITIONED|
                    project ([$$0])
                    -- STREAM_PROJECT  |PARTITIONED|
                      exchange
                      -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                        data-scan []<-[$$9, $$0] <- test:Orders
                        -- DATASOURCE_SCAN  |PARTITIONED|
                          exchange
                          -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                            empty-tuple-source
                            -- EMPTY_TUPLE_SOURCE  |PARTITIONED|
              exchange
              -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                project ([$$10, $$13])
                -- STREAM_PROJECT  |PARTITIONED|
                  assign [$$13] <- [function-call: asterix:field-access-by-ind
ex, Args:[%0->$$1, AInt32: {1}]]
                  -- ASSIGN  |PARTITIONED|
                    exchange
                    -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                      data-scan []<-[$$10, $$1] <- test:Customers
                      -- DATASOURCE_SCAN  |PARTITIONED|
                        exchange
                        -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                          empty-tuple-source
```

Figure 17: Optimized Query Plan for Block Nested Loops Join in AsterixDB

Table 2 summarizes details of some of the common use cases that each of the join methods that we discussed is suitable for,

Table 2: Use cases that apply to different Join methods.

Join Algorithm	**Use case**
Hybrid Hash Join	No index and or no-order (R.attrA = S.attrB)
Indexed nested loops join	Small R (outer relation) and indexed S (outer relation) (R.attrA /*+indexnl */ = S.attrB)
Block nested loops join	Non-equality based joins (R.attrA > S.attrB)

4.2 Aggregates

Aggregation is a very important statistical concept to summarize information about large amounts of data. The idea is to represent a set of items by a single value or to classify items into groups and determine one value per group. There are two forms of aggregations, called scalar aggregates and aggregate functions. Scalar aggregates calculate a single scalar value from a unary input relation, e.g., the sum of the salaries of all employees. Scalar aggregates can easily be determined using a single pass over a data set. Some systems exploit indices, in particular to calculate minimum, maximum, and count. Aggregate functions, on the other hand, determine a set of values from a binary input relation, e.g., the sum of salaries for each department. Aggregate functions are relational operators, i.e., they consume and produce relations [13]. Parallel algorithms for aggregation are best divided into a local step and a global step. First, aggregates are calculated locally at each node, and then data is aggregated globally at one of the chosen nodes. For aggregation, local and global aggregate functions may

differ. For example, to perform a global count, the local aggregation counts while the global aggregation sums local counts into a global count [13].

```
// DDL to create an open type
create type EmpType
as open {
        id: int32,
        name: string,
        dname: string
}

// DDL to create Employee dataset
create dataset Employee(EmpType) primary key id;

// Insert data into Employee dataset.

// Aggregate Query that performs a count of all employees
count(for $l in dataset Employee return $l)
```

Figure 18: Aggregate Query that returns count of employees in AsterixDB.

To illustrate an aggregate count function in AsterixDB, consider the example in Figure 18, that performs a count of all the employees in the Employee dataset.

In Figure 19, the optimized query plan for the AQL aggregate count query in Figure 18 is illustrated. A data source scan is performed on the Employee dataset. A project is then done to retain only the required attributes of the Employee dataset, which is followed by a local aggregation of data at each of the nodes that counts the tuples on each node. A random merge exchange is then performed. A global aggregate, sum, is applied to the incoming data received from the exchange and the results are

returned.

```
distribute result [%0->$$4]
-- DISTRIBUTE_RESULT  |UNPARTITIONED|
  exchange
  -- ONE_TO_ONE_EXCHANGE  |UNPARTITIONED|
    aggregate [$$4] <- [function-call: ASTERIX_PRIVATE:agg-sum, Args
:[%0->$$6]]
    -- AGGREGATE  |UNPARTITIONED|
      exchange
      -- RANDOM_MERGE_EXCHANGE  |PARTITIONED|
        aggregate [$$6] <- [function-call: ASTERIX_PRIVATE:agg-count
, Args:[%0->$$0]]
        -- AGGREGATE  |PARTITIONED|
          project ([$$0])
          -- STREAM_PROJECT  |PARTITIONED|
            exchange
            -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
              data-scan []<-[$$5, $$0] <- test:Employee
              -- DATASOURCE_SCAN  |PARTITIONED|
                exchange
                -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                  empty-tuple-source
                  -- EMPTY_TUPLE_SOURCE  |PARTITIONED|
```

Figure 19: Optimized Logical Plan for count aggregate query in AsterixDB

4.3 Group By

AsterixDB internally utilizes both sorting and hashing techniques to achieve efficient group-by performance. By choosing among sort-based, hash-based, and hash-sort-hybrid strategies, AsterixDB can reach good performance on both I/O and CPU costs for grouping huge data with different distributions. Specifically, the sort-based grouping algorithm sorts the input data based on the grouping key and finishes the group-by through a scan over the sorted data to produce running group-by results. This strategy

works best when the data has been pre-sorted on the group-by keys, or when the output of the group-by is required to be sorted. The hash-based algorithm uses a hybrid-hash strategy to fully utilize the memory for group-by, and it also partitions the input data if they cannot be fit into memory; the spilled partitions in the hash-based algorithm can be processed recursively. The Hash-based strategy is the best choice in most of the group-by scenarios when the data distribution is not highly skewed. Finally the hash-sort-hybrid algorithm [57] uses both hashing (to do grouping as early as possible in order to save I/O) and sorting (to avoid deep recursion if the data is highly skewed), and it works best for skewed data. The working of parallel group by in AsterixDB is shown in Figure 20. Figure 21 illustrates a simple AQL query that groups employees by department and returns a count of the number of employees per department.

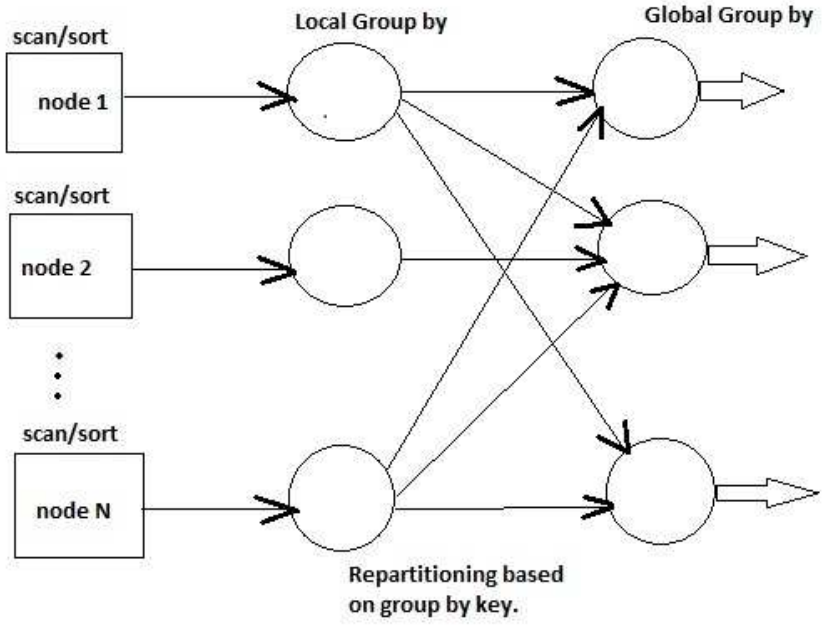

Figure 20: Parallel Group by in AsterixDB

```
// DDL to create an open type
create type Emp
as open {
        id: int32,
        name: string,
        dname: string
}

// DDL to create Employee dataset
create dataset Employee(Emp) primary key id;

// Insert data into Employee dataset.

// AQL Query that performs grouped aggregation.
for $l in dataset Employee
let $x := $l.id
group by $m := $l.dname with $x
return { "dname" : $m, "emp-count": count($x) }
```

Figure 21: Grouped Aggregation in AsterixDB.

Figure 22 illustrates an optimized query plan for the group by query in Figure 21. A data source scan of the dataset is performed and a projection is applied to the input data so that only the required attributes are retained. A sort of the data is done because the pre-clustered group by operator requires data to be sorted before grouping. In the pre-clustered group by step the group by operator gets the local aggregation results since it is run on each local node. After the local group-by, the partial group-by results are hash-partitioned and distributed to different nodes so that all results about a given grouping key are all on a same node. Note that here a hash_partition_merge_exchange_merge connector is used; this connector ensures that records are hash partitioned, and on the receiver side records are still sorted. Another group-by is then applied on each node after the hash distribution. Lastly, the global group by results are fed to a project operator which returns only the required attributes and the results are returned.

```
distribute result [%0->$$8]
-- DISTRIBUTE_RESULT  |PARTITIONED|
  exchange
  -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
    project ([$$8])
    -- STREAM_PROJECT  |PARTITIONED|
      assign [$$8] <- [function-call: asterix:closed-record-constructor, Args:
[AString: {dname}, %0->$$2, AString: {emp-count}, %0->$$12]]
      -- ASSIGN  |PARTITIONED|
        exchange
        -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
          group by ([$$2 := %0->$$14]) decor ([]) {
                    aggregate [$$12] <- [function-call: asterix:agg-sum, Args:
[%0->$$13]]
                    -- AGGREGATE  |LOCAL|
                      nested tuple source
                      -- NESTED_TUPLE_SOURCE  |LOCAL|
                  }
          -- PRE_CLUSTERED_GROUP_BY[$$14]  |PARTITIONED|
            exchange
            -- HASH_PARTITION_MERGE_EXCHANGE MERGE:[$$14(ASC)] HASH:[$$14]  |P
ARTITIONED|
              group by ([$$14 := %0->$$10]) decor ([]) {
                        aggregate [$$13] <- [function-call: asterix:agg-count,
 Args:[AInt32: {1}]]
                        -- AGGREGATE  |LOCAL|
                          nested tuple source
                          -- NESTED_TUPLE_SOURCE  |LOCAL|
                       }
              -- PRE_CLUSTERED_GROUP_BY[$$10]  |PARTITIONED|
                exchange
                -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                  order (ASC, %0->$$10)
                  -- STABLE_SORT [$$10(ASC)]  |PARTITIONED|
                    exchange
                    -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                      project ([$$10])
                      -- STREAM_PROJECT  |PARTITIONED|
                        assign [$$10] <- [function-call: asterix:field-access-
by-index, Args:[%0->$$0, AInt32: {2}]]
                        -- ASSIGN  |PARTITIONED|
                          project ([$$0])
                          -- STREAM_PROJECT  |PARTITIONED|
                            exchange
                            -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                              data-scan []<-[$$11, $$0] <- test:Employee
                              -- DATASOURCE_SCAN  |PARTITIONED|
                                exchange
                                -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                                  empty-tuple-source
                                  -- EMPTY_TUPLE_SOURCE  |PARTITIONED|
```

Figure 22:Optimized Logical Plan for a grouped aggregation query in AsterixDB

4.4 Order By

The order by clause is used in database systems to organize data in a certain order, either ascending or descending, as specified by the user, or to provide data in sorted order to another operator in the pipeline as required. In a parallel shared-nothing database system with N nodes (machines), each of those N nodes performs the sort on its local data; later all these nodes can send their data, which is sorted, to a node where a merge is performed on the data received from the different nodes; the data after the merge operation is in the required sorted order. It must be noted that sorting data is an expensive operation, and it can slow down the performance of certain queries significantly, ie., if sorting is performed where it is not required to be performed. Different techniques for implementing sorting in database management systems are covered in [55]

In Figure 23, we see the implementation of the order by clause at a high-level in AsterixDB. Each node performs a local sort and then a merge is performed on the data received from each of the nodes at one of the available nodes. Data is then available in the required sorted order, after the merge, for processing by other operators.

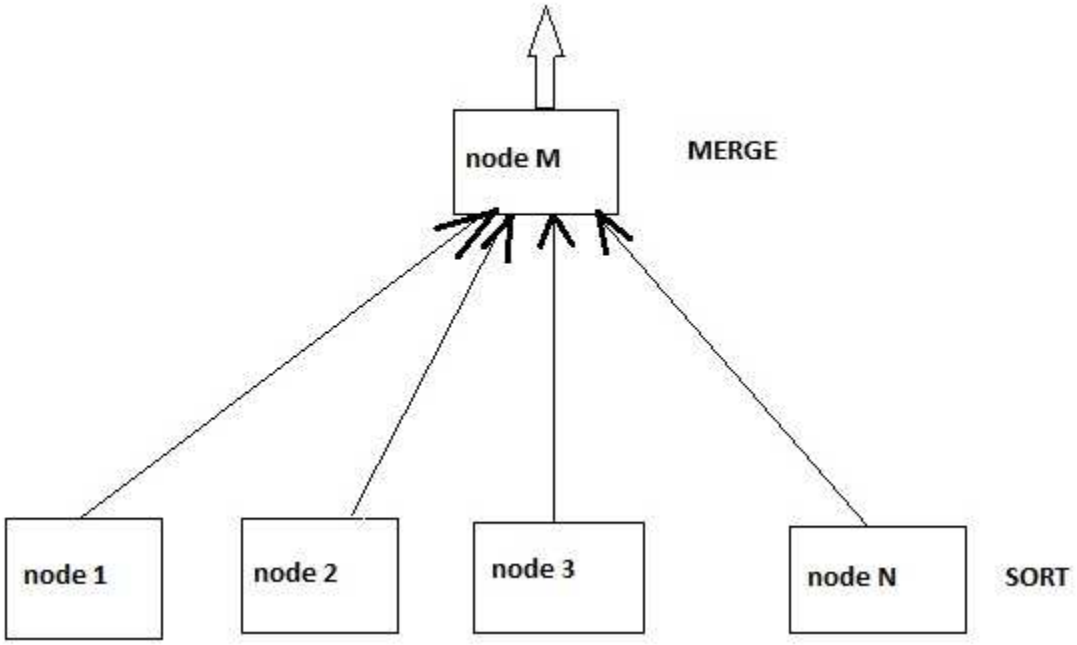

Figure 23: Order by : Sort at each node and Merge at another node.

```
// DDL statement to create an open type
create type Emp
as open {
        id: int32,
        name: string
}

// DDL to create Employee dataset
create dataset Employee(Emp) primary key id;

// Insert data into Employee dataset.

// AQL Query that orders results
// in ascending order
for $l in dataset Employee
order by $l.name
return $l
```

Figure 24: AQL Query that performs ordering using Order by clause

In Figure 24, an AQL query that performs ordering of results using the order by clause is illustrated. The corresponding optimized query plan is available in Figure 25. A data source scan is performed on the Employee dataset, a project is performed to retain only the required attribute “name”, followed by a sort operation that sorts data in ascending order on each of the nodes. A global merge is then effected by the sort merge exchange connector on the sorted data from all nodes at one of the available nodes. The project operator later projects only the required attribute and then results are returned in ascending order.

```
distribute result [%0->$$0]
-- DISTRIBUTE_RESULT  |PARTITIONED|
  exchange
  -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
    project ([$$0])
    -- STREAM_PROJECT  |PARTITIONED|
      exchange
      -- SORT_MERGE_EXCHANGE [$$4(ASC) ]  |PARTITIONED|
        order (ASC, %0->$$4)
        -- STABLE_SORT [$$4(ASC)]  |PARTITIONED|
          exchange
          -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
            assign [$$4] <- [function-call: ASTERIX_PRIVATE:field-access-by-in
dex, Args:[%0->$$0, AInt32: {1}]]
            -- ASSIGN  |PARTITIONED|
              project ([$$0])
              -- STREAM_PROJECT  |PARTITIONED|
                exchange
                -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                  data-scan []<-[$$3, $$0] <- test:Employee
                  -- DATASOURCE_SCAN  |PARTITIONED|
                    exchange
                    -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                      empty-tuple-source
                      -- EMPTY_TUPLE_SOURCE  |PARTITIONED|
```

Figure 25: Optimized Query Plan for an Order by query in AsterixDB

4.5 Limit

Limit is applied to restrict the the number of tuples returned by a query. Limits can be used in certain use-cases where the user is interested in knowing only the top N results, for example, the employees with the top ten highest salaries. To achieve this we can do an order by on the salary attribute of the Employee dataset, in descending order, and return only the first ten tuples, which will give us the desired result. It should be noted that limits must eventually be applied globally in a parallel environment, rather than calculating limits just locally at each of the sites.

Figure 26 illustrates an AQL query that applies a limit simply to restrict the number of tuples that are returned by a select query. In Figure 27, we see the optimized query plan for the query. After a data source scan on each site, a local sort in ascending order is performed and a local limit is applied to the data returned by the local sort. A sort merge exchange connector is used to merge the data in sorting order and combine the tuples on one site. We again apply the global limit on the aggregated tuples to pick the first N tuples and the results are returned.

```
// DDL to create an open type.
create type EmpType
as open {
        id: int32,
        name: string,
        salary: int32
}
// DDL to create Employee dataset.
create dataset Employee(EmpType) primary key id;

// AQL Query that applies a limit to restrict number of results returned.
for $l in dataset Employee
order by $l.salary
limit 5
return $l
```

Figure 26: AQL Query that restricts query results using limit clause in AsterixDB

```
distribute result [%0->$$0]
-- DISTRIBUTE_RESULT  |UNPARTITIONED|
  exchange
  -- ONE_TO_ONE_EXCHANGE  |UNPARTITIONED|
    limit AInt32: {5}
    -- STREAM_LIMIT  |UNPARTITIONED|
      project ([$$0])
      -- STREAM_PROJECT  |PARTITIONED|
        exchange
        -- SORT_MERGE_EXCHANGE [$$4(ASC) ]  |PARTITIONED|
          limit AInt32: {5}
          -- STREAM_LIMIT  |PARTITIONED|
            exchange
            -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
              order (ASC, %0->$$4)
              -- STABLE_SORT [$$4(ASC)]  |PARTITIONED|
                exchange
                -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                  assign [$$4] <- [function-call: ASTERIX_PRIVATE:field-access
-by-index, Args:[%0->$$0, AInt32: {2}]]
                  -- ASSIGN  |PARTITIONED|
                    project ([$$0])
                    -- STREAM_PROJECT  |PARTITIONED|
                      exchange
                      -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                        data-scan []<-[$$3, $$0] <- test:Employee
                        -- DATASOURCE_SCAN  |PARTITIONED|
                          exchange
                          -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                            empty-tuple-source
                            -- EMPTY_TUPLE_SOURCE  |PARTITIONED|
```

Figure 27: Optimized Logical plan for an AQL query that applies a limit

Chapter 5

Strategic Test Plan

In this chapter we discuss the test strategies used to verify and validate the correctness of AsterixDB's parallel query processing functionalities (eg. joins, group by, aggregates, order by, and limits). This chapter is divided into five sections, and each section focuses on the tests for the functionality that that section is intended to cover. All tests described in each of the sections are first executed on a standalone setup (meaning one Cluster Controller and one Node Controller) on a single machine and then executed on a clustered environment where we have one Cluster Controller and one or more Node Controllers running on the cluster. Each section gives a high level description of the tests, and a reference to the low-level tests written in AQL (the Asterix Query Language) is provided in the Appendix section of this thesis for reference. All tests are executed via AsterixDB's existing HTTP-based API test framework. This chapter does not provide a comprehensive test plan that covers every area that can be tested; however, we attempt to verify and validate certain interesting features of the parallel query processing abilities of AsterixDB to the maximum extent possible.

AsterixDB uses a HTTP-based test framework that in turn uses AsterixDB's REST API to submit AQL queries (i.e., DDL/DML statements) for execution by the engine. DDL and DML statements are submitted via independent sessions to the engine for execution. The results returned by the system after query execution are verified by comparing the actual results received with expected results which are available in an

expected result file for each test case. The test framework supports execution of tests on a single node and on multi-node environments. AsterixDB has a result distribution framework which aggregates results from all the partitions of the cluster and returns a single stream of results, that are compared with the expected results. The test framework also supports execution of negative tests. An expected Exception message and Errors can be specified, for each negative test, for the framework to verify if the actual Exception or error message is the same as the expected Exception or error message.

5.1 Joins

5.1.1 Hybrid Hash Join

Equi-joins in AsterixDB are implemented using the Hybrid Hash Join method, which was discussed in section 4.1.1 of this thesis. The focus of tests in this section is towards testing the equi-join for different distributions of data across the nodes, using a join predicate which is equality based, like R.attrA = S.attrB. The tests cover different scenarios based on the definition of the two join key attributes of each of the datasets R and S. To test different combinations of the join key attributes in an equi-join predicate, attributes A and B from R and S can be defined as primary key, or they can be a non key or they may be part of a two-part composite key. Query plans are validated for “correctness”; one very important verification from the query plan is to verify that only that dataset whose join key attribute is *not* defined as a primary key is repartitioned and sent over to the other dataset. Table 3 lists the different test scenarios for Hybrid Hash

Join in AsterixDB. All test scenarios listed in there use two internal datasets which are hash partitioned on their primary key.

Table 3: Hybrid Hash Join Tests

	Aim of Test & Description	**Predicate used to join two datasets**	**Expected Results**
1.	Test to verify that there is no re-partitioning of data from the two datasets, since both datasets are hash partitioned on their primary key. Equijoin wherein T1.ID and T2.ID are primary keys	for $m in dataset T1 for $n in dataset T2 where $m.ID = $n.ID return {“m”:$m,”n”:$n}	Verify that the join is a Hybrid Hash join and verify correctness of results returned. Also verify that we do not do a hash partition exchange, as both join attributes are defined as the primary key in its dataset.
2.	Equijoin test , in which T2.ID is NOT defined as a primary key, and T1.ID is defined as a primary key.	for $m in dataset T1 for $n in dataset T2 where $m.ID = $n.ID return {“m”:$m,”n”:$n}	Verify that the join is a Hybrid Hash join and verify correctness of results returned. Also verify that only dataset T2 is hash partition exchanged and sent to T1.
3.	Equijoin test, in which T1.ID is NOT defined as a primary key and T2.ID is defined as a primary key.	for $m in dataset T1 for $n in dataset T2 where $m.ID = $n.ID return {“m”:$m,”n”:$n}	Verify that the join is a Hybrid Hash join and verify correctness of results returned. Also verify that only dataset T1 is hash partition exchanged and sent to T2.
4.	Equijoin test in which ID is not defined as primary key in both datasets T1, T2 respectively. Note : The data used in this test will include some duplicates.	for $c in dataset customers for $o in dataset orders where $c.id = $o.id return {"cust-id":$c.id,"shipToName":$o.shipToName}	Verify that the join is a Hybrid Hash join and verify correctness of results returned. Also verify that both the datasets are repartitioned, as both the join key attributes are not defined as primary keys.
5.	Equijoin test in which T1.ID and T2.ID are defined as PK T2.name is secondary index. Note : The residual name	for $l in dataset t1 for $m in dataset t2 where $l.id=$m.id and $m.name = "John Doe" return {"t1":$l,"t2":$m}	Verify that the join is a Hybrid Hash join and verify correctness of results returned. Also verify that there is no repartitioning of either of

	predicate must be verified by the presence of index scan in the query plan.		the datasets from the query plan.
6.	Equijoin test where join predicate is of the form : *expression1* **and** *expression2* Customers.ID is PK Orders.ID is PK customer.name and orders.shipToName are non key attributes	for $l in dataset Customers for $m in dataset Orders where $l.id=$m.id and $l.name=$m.shipToName return {"cid":$l.id,"ship-to-name":$m.shipToName}	Verify that the join is a Hybrid Hash join and verify correctness of results returned. Also verify that there is no repartitioning of either of the datasets from the query plan.
7.	Equijoin test where join key attributes are defined as two-part primary keys in the datasets. Customer(id1,id2) is PK Orders(id1,id2) is PK	for $l in dataset Customers for $m in dataset Orders where $l.id1=$m.id1 and $l.id2 = $m.id2 return {"customer":$l,"orders":$m}	Verify that the join is a Hybrid Hash join and verify correctness of results returned. Also verify that there is no re-partitioning of either of the datasets, from the query plan.
8.	Equijoin test, where the join key attributes in the join predicate are a prefix of a two-part primary key. Customers(id1,id2) is PK Orders(id1,id2) is PK	for $l in dataset Customers for $m in dataset Orders where $l.id1=$m.id1 return {"customer":$l,"orders":$m}	Verify that the join is a Hybrid Hash join and verify correctness of results returned. Also verify from the query plan that both datasets are re-partitioned.
9.	Same as case 8, except that join key attributes are a suffix of the key. Customers(id1,id2) is PK Orders(id1,id2) is PK	for $l in dataset customer for $m in dataset orders where $l.id2 = $m.id2 return {"customer":$l,"orders":$m}	Verify that the join is a Hybrid Hash join and verify correctness of results returned. Also verify from the query plan that both the datasets are re-partitioned.
10.	Equijoin on internal datasets T1 and T2, T1 is defined as open type and T2 is defined as closed type. (fname,lname) are two part primary keys defined in T1 and T2 Note : T1 has additional field/data in it, meaning fields that were not defined in the type.	for $m in dataset T1 for $n in dataset T2 where $m.fname = $n.fname and $m.lname = $n.lname return {“t1”:$m, “t2”:$n}	Verify that join is a Hybrid Hash join and verify correctness of results returned. Also verify that neither of the datasets is re-partitioned.

11	Equijoin on internal datasets T1 and T2, both T1 and T2 are defined as open. The join predicate involves undefined field from T1 and T2. Note : “city” field of type string, is not defined in the type for T1 or T2.	for $m in dataset T1 for $n in dataset T2 where $m.city = $n.city return {“t1”:$m,”t2”:$n}	Verify that the join is a Hybrid Hash join and verify correctness of results returned.
12	Equijoin on internal datasets T1 and T2, both T1 and T2 are defined as open. The join predicate involves undefined field from T1 and T2. Note : “zip” field of type integer, is not defined in the type.	for $m in dataset T1 for $n in dataset T2 where $m.zip = $n.zip return {“t1”:$m,”t2”:$n}	Verify that the join is a Hybrid Hash join and verify correctness of results returned.
13.	Equijoin on internal datasets T1 and T2, both T1 and T2 are defined as open. The join predicate is over an undefined field from T1 and T2. Note : “hobbies” field in **not defined** in the type. “hobbies” here is an unordered list of strings.	for $m in dataset T1 for $n in dataset T2 for $a in $m.hobbies for $b in $n.hobbies where $a = $b return {“t1”:$m,”t2”:$n}	Verify that the join is a Hybrid Hash join and verify correctness of results returned.
14.	Equijoin on internal datasets T1 and T2, both T1 and T2 are defined as open. The join predicate is defined over values returned by iterating over an unordered list from T1 and T2. Note: “hobbies” field is **defined** in the type as an unordered list {{ string }}	for $m in dataset T1 for $n in dataset T2 for $a in $m.hobbies for $b in $n.hobbies where $a = $b return {“t1”:$m,”t2”:$n}	Verify that the join is a Hybrid Hash join and verify correctness of results returned.
15.	Equijoin that involves a three-part composite key (fname, mi, lname) defined as primary key in datasets T1 and T2	for $m in dataset T1 for $n in dataset T2 where ($m.fname=$n.fname and $m.mi=$n.mi and $m.lname=$m.lname) return {“m”:$m,”n”:$n}	Verify that the join is a Hybrid Hash join and verify correctness of results returned. Also verify that neither of the datasets is repartitioned.
16.	Test to verify Left Outer Join (Equi-join) PK = PK Customers.ID is defined as primary key Orders.ID is defined as	for $l in dataset Customers return { "cust":$l, "orders": for $m in dataset Orders where $l.id=$m.id return $m	Verify that the left outer join is a Hybrid Hash join and verify correctness of results returned. Also verify that neither of the datasets is repartitioned.

	primary key	}	

17.	Test to verify Left Outer Join (Equi-join)
	PK = Non-Key
	
	Customers.ID is defined as primary key
	Orders.ID is NOT defined as primary key
		for $l in dataset Customers
		return {
		"cust":$l,
		"orders": for $m in dataset Orders
		where $l.id=$m.id
		
		return $m
		}
			Verify that the left outer join is a Hybrid Hash join and verify correctness of results returned. Also verify that Orders dataset is repartitioned, because Order.id is not defined as a primary key.

18.	Test to verify Left Outer Join (Equi-join)
	Non-Key = PK
	
	Customers.ID is NOT defined as primary key
	Orders.ID is defined as primary key
		for $l in dataset Customers
		return {
		"cust":$l,
		"orders": for $m in dataset Orders
		where $l.id=$m.id
		
		return $m
		}
			Verify that the left outer join is a Hybrid Hash join and verify correctness of results returned. Also verify that Customers dataset is repartitioned, because Customers.id is not defined as a primary key.

19.	Test to verify Left Outer Join (Equi-join)
	Non-Key = Non-Key
	
	Both Customers.name and Orders.shipToName are not defined as primary key.
		for $l in dataset Customers
		return {
		"cust":$l,
		"orders": for $m in dataset Orders
		where $l.name=$m.shipToName
		
		return $m
		}
			Verify that the left outer join is a Hybrid Hash join and verify correctness of results returned. Verify that both Orders and Customers are repartitioned as the join key attributes are not defined as primary keys.

20	Test to verify that join predicate of the form, expression(R.attrA) = expression(S.attrB) results in a hybrid hash join.	for $c in dataset DsOne for $o in dataset DsTwo where fun-1(R.attrA) = fun-2(S.attrB) return { “attrA” : $c.attrA , “attrB” : $o.attrB }	Verify that the join is a Hybrid Hash join.

5.1.2 Indexed Nested Loops Join

In this section we discuss the tests that verify the correctness of indexed nested loops join in AsterixDB. Indexed nested loops join is discussed in section 4.1.2 of this thesis.

The focus of these tests is to verify that the outer relation is broadcast (or hash partitioned, in the primary key case) to every other node, and that the secondary index defined on the join key attribute of the inner relation is used to do an index lookup to find a match for the value of the join key attribute of the outer relation. Table 4 lists the different tests that have been written and executed to test indexed nested loops joins in AsterixDB. Each row in Table 4 actually corresponds to five different tests, as each of the scenarios listed in the table is tested using all five forms of the join predicate as listed below. A join predicate with a hint to do an index scan (B+ Tree index) lookup on the inner relation to find the match can be of any one of the following forms;

i) where $l.<*attribute*> **/*+ indexnl */ =** $m.<*attribute*>
ii) where $l.<*attribute*> **/*+ indexnl */ <** $m.<*attribute*>
iii) where $l.<*attribute*> **/*+ indexnl */ >** $m.<*attribute*>
iv) where $l.<*attribute*> **/*+ indexnl */ <=** $m.<*attribute*>
v) where $l.<*attribute*> **/*+ indexnl */ >=** $m.<*attribute*>

Table 4: Indexed Nested Loops Join Tests

	Aim of Test and description	**Expected Result**
1.	Test to verify that secondary index defined on Orders.shipToName is picked to find a match for the Customers.name attribute, in Orders dataset. A secondary index is defined on Orders.shipToName. for $o in dataset Orders for $c in dataset Customers where $c.name /* +indexnl */ = $o.shipToName return {"name":$o.shipToName}	Verify that a secondary index scan is performed on the Orders dataset and verify the correctness of results returned.
2.	Test to verify that secondary index defined on Orders.cid is picked to find a match for the Customers.cid attribute, in Orders dataset.	Verify that a secondary index scan is performed on the Orders dataset and that Orders dataset is broadcast exchanged

	A secondary index is defined on Orders.cid for $o in dataset Orders for $c in dataset Customers where $c.cid /* +indexnl */ = $o.cid return {"name":$o.shipToName}	and verify the correctness of results returned.
3.	Test to verify that the primary index defined on Customers.cid is picked to find a match for Orders.cid Customers.cid is the primary index defined on Customers dataset. for $o in dataset Orders for $c in dataset Customers where $o.cid /* +indexnl */ = $c.cid return {"name":$c.name}	Verify that the tuples of Orders dataset are Hash partitioned because we know that Orders is hash partitioned on its primary key, and that we are probing Customers dataset's primary index Customers.cid in the join. IMPORTANT : Verify from the query plan that dataset Orders is not broadcast exchanged!
4.	Test to verify that an indexed nested loops join is NOT performed when the join key attribute of the inner relation is not defined as a primary or secondary index. Instead, AsterixDB chooses to perform a Hybrid Hash Join, ignoring the hint. Customers.name is not defined as primary or secondary index in Customers dataset. for $o in dataset Orders for $c in dataset Customers where $o.shipToName /* +indexnl */ = $c.name return {"name":$c.name}	Verify from the query plan that a Hybrid Hash join is performed and also verify that there is no B+ Tree index scan in the query plan. Verify the correctness of results returned.

5.	Test to verify that composite secondary index defined on attributes of inner relation DsTwo is picked in the join.
	
	create type TypeOpen as open {
	id: int32,
	fname: string,
	lname: string
	}
	create dataset DsOne(TypeOpen) primary key id;
	create dataset DsTwo(TypeOpen) primary key id;
	
	create index indx-dst on DsTwo(fname,lname);
		Verify from the query plan that there is a broadcast exchange of dataset DsOne, since DsTwo's join key attribute is not its primary index and verify that we probe the secondary index defined on DsTwo.

	//insert data into datasets for $o in dataset DsOne for $t in dataset DsTwo where $o.fname /* +indexnl */ = $t.fname and $o.lname /* +indexnl */ = $t.lname return {"name":$t.lname}	

6.	Test to verify that prefix of a composite secondary index defined on attributes of inner relation DsTwo is picked in the join.
	
	create type TypeOpen as open {
	id: int32,
	fname: string,
	lname: string
	}
	create dataset DsOne(TypeOpen) primary key id;
	create dataset DsTwo(TypeOpen) primary key id;
	
	create index indx-dst on DsTwo(fname,lname);
	
	//insert data into datasets
	
	// Only prefix of two part secondary key is used.
	for $o in dataset DsOne
	for $t in dataset DsTwo
	where $o.fname /* +indexnl */ = $t.fname
	return {"name":$t.lname}
		Verify that an indexed nested loops join is performed and verify correctness of results.
		
		Note: This is currently not supported, an enhancement request is reported to support this in the future.

5.1.3 Nested Loops Join

This section describes the tests that verify the correctness of nested loops join in AsterixDB. All tests listed in Table 5 are written and executed on a standalone and a cluster setup. Nested loops joins in AsterixDB are discussed in section 4.1.3 of this thesis.

Table 5: Nested Loops Join Tests

	Test Description and Aim of Test case.	**Predicate used to join two internal datasets**	**Expected Results**
1.	NL join test, with join predicate of the form : t1.attribute-a > t2.attribute-b	for $m in dataset customers for $n in dataset orders where $m.id > $n.id return {“cust-name”:$m.name,”order-id”:$n.id}	Verify correctness of results and that this is a nested loop join.
2.	NL join test, with join predicate of the form : t1.attribute-a >= t2.attribute-b	for $m in dataset customers for $n in dataset orders where $m.id >= $n.id return {“cust-name”:$m.name,”order-id”:$n.id}	Verify correctness of results and that this is a nested loop join.
3.	NL join test, with join predicate of the form : t1.attribute-a < t2.attribute-b	for $m in dataset customers for $n in dataset orders where $m.id < $n.id return {“cust-name”:$m.name,”order-id”:$n.id}	Verify correctness of results and that this is a nested loop join.
4.	NL join test, with join predicate of the form : t1.attribute-a <= t2.attribute-b	for $m in dataset customers for $n in dataset orders where $m.id <= $n.id return {“cust-name”:$m.name,”order-id”:$n.id}	Verify correctness of results and that this is a nested loop join.
5.	NL join test, with join predicate of the form : t1.attribute-a != t2.attribute-b Anti-Join wherein T1.ID and T2.ID are defined as primary keys	for $m in dataset T1 for $n in dataset T2 where $m.ID != $n.ID return {“m”:$m,”n”:$n}	Verify correctness of results and verify that this is a nested loop join in the optimized query plan.
6.	NL join test, with join predicate of the form : t1.attribute-a > t2.attribute-a and t1.attribute-b < t2.attribute-b	for $l in dataset t1 for $m in dataset t2 where $l.name > $m.name and $l.id < $m.id return {"l":$l,"m":$m}	Verify correctness of results and verify that this is a nested loop join in the optimized query plan.
7.	NL join test, with join predicate of the form :	for $l in dataset t1 for $m in dataset t2 where $l.name >= $m.name	Verify correctness of results and verify that this is a nested loop join from the query plan.

	t1.attribute-a >= t2.attribute-a and t1.attribute-b <= t2.attribute-border	and $l.id <= $m.id return {"l":$l,"m":$m}	
8.	NL join test, with join predicate of the form : t1.attribute-a > t2.attribute-a and t1.attribute-b <= t2.attribute-border	for $l in dataset t1 for $m in dataset t2 where $l.id > $m.id and $l.id <= $m.id return {"l":$l,"m":$m}	Verify correctness of results and verify that this is a nested loop join from the query plan.
9.	NL join test, with join predicate of the form : t1.attribute-a >= t2.attribute-a and t1.attribute-b < t2.attribute-border	for $l in dataset t1 for $m in dataset t2 where $l.id > $m.id and $l.id <= $m.id return {"l":$l,"m":$m}	Verify correctness of results and verify that this is a nested loop join from the query plan.

5.2 Aggregates

In this section we verify the correctness of different aggregate functions in AsterixDB. Aggregates in AsterixDB are discussed in section 4.2 of this thesis. The focus of the tests in this section is to ensure that the aggregate functions perform the local and global aggregations as designed, and that they handle inputs of different datatypes and also handle nulls correctly. All tests discussed in this section were executed on both a standalone setup and on a cluster of nodes. Table 6 summarizes different inputs (primitive types) that will be passed to the aggregate functions listed in column one of the table. The fields of the primitive types mentioned in Table 6 are defined as **mandatory fields** (not optional fields) of the internal dataset over which each of the aggregate functions is executed. Each test will be a scan query over an internal dataset without any predicates.

All tests discussed in Table 6 will then be repeated for fields that are defined as **optional fields** in the internal dataset over which each of the aggregate functions is executed. Some or all of the tuples may have nulls for those optional fields. Each test will again be a scan query over the internal dataset without any predicate.

Table 6: Different types of inputs from data stored in datasets to test aggregate functions.

function name	int	float	double	string	time	date	datetime	duration	interval
sum()	Yes	Yes	Yes	NA	Yes	Yes	Yes	Yes	Yes
count()	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
avg()	Yes	Yes	Yes	NA	Yes	Yes	Yes	Yes	Yes
max()	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
min()	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

Note that the fields that hold data are mandatory.
Yes : indicates that feature is supported and will be tested.
NA : indicates that feature is not supported and will not be tested

Tests for different uses of aggregate functions in AsterixDB and AQL can be classified into three categories: Simple aggregates over a dataset, Grouped aggregates over a dataset, and Simple aggregates over nested collections. In all these tests, a schema similar to the UserType schema shown below will be used:

```
create type UserType
as open {
        name: string,
        age: int64,
        kids-ages: {{int8}},
        kind: string,
        hobbies: {{string}}
}
create dataset Users(UserType) primary key age;
```

5.2.1. Simple aggregates over a dataset

Basic tests to verify the correctness of simple aggregates were described in detail in Table 6. An additional set of tests, similar to those discussed in Table 6, use the UserType schema. These tests are :

i) Test to get the average age of Users, using the avg() aggregate function.

ii) Test to get the count of all Users, using the count() aggregate function.

iii) Test to get the maximum age of a User, using the max() aggregate function.

iv) Test to get the minimum age of a User, using the min() aggregate function.

5.2.2 Grouped aggregates over dataset

Section 5.3 of this thesis discusses the tests for grouped aggregate queries in AQL.

5.2.3 Simple aggregates over a nested collection

i) Test to get the names of the Users and their average kids-age

ii) Test to get the names of the Users and the counts of their hobbies.

5.3 Group By

This section covers tests that will be executed to verify the serial and parallel correctness of the group by clause and its application to perform grouped aggregation. Group by functionality in AsterixDB was discussed in section 4.3 of this thesis.

5.3.1. Tests to cover group by key

The group by key(s) can be of any of the primitive data types that AsterixDB supports. All primitive types will be used as the group by key(s) in tests to perform the grouping. These tests are described in Table 7. All tests in Table 7 will use the schema AllType and dataset definition DsOne as shown below:

```
create type AllType
as open {
    id : int32,
    int_m : int32,
    int_o : int32?,
    int_64_m : int64,
    string_m : string,
    string_o : string?,
    tm_m: time,
    dt_m: date,
    dt_tim_m: datetime,
    intrvl: interval,
    pt_m: point,
    flt_m: float
  };

create dataset DsOne(AllType) primary key id;
```

Table 7 : Tests to verify group by key use on all primitive types

	Test Description & Aim	**Expected Result**
1.	Test to group results based on a mandatory field of type int32 for $x in dataset DsOne let $id := $x.id group by $m := $x.int_m with $id return { "group-by-key": $m, "group-members": $id }	Verify that the results returned are grouped by the group by key int_m
2.	Test to group results based on an optional field of type int32 for $x in dataset DsOne let $id := $x.id group by $m := $x.int_o with $id	Verify that the results returned are grouped by the group by key int_o

	return { "group-by-key": $m, "group-members": $id }	
3.	Test to group results based on a mandatory field of type string for $x in dataset DsOne let $id := $x.id group by $m := $x.string_m with $id return { "group-by-key": $m, "group-members": $id }	Verify that the results returned are grouped by the group by key string_m
4.	Test to group results based on an optional field of type string for $x in dataset DsOne let $id := $x.id group by $m := $x.string_o with $id return { "group-by-key": $m, "group-members": $id }	Verify that the results returned are grouped by the group by key string_o
5.	Test to group results based on a mandatory field of type date for $x in dataset DsOne let $id := $x.id group by $m := $x.dt_m with $id return { "group-by-key": $m, "group-members": $id }	Verify that the results returned are grouped by the group by key dt_m
6.	Test to group results based on an mandatory field of type datetime for $x in dataset DsOne let $id := $x.id group by $m := $x.dt_tim_m with $id return { "group-by-key": $m, "group-members": $id }	Verify that the results returned are grouped by the group by key dt_tim_m
7.	Test to group results based on a mandatory field of type interval for $x in dataset DsOne let $id := $x.id group by $m := $x.intrvl with $id return { "group-by-key": $m, "group-members": $id }	Verify that the results returned are grouped by the group by key intrvl
8.	Test to group results based on an mandatory field of type float.	Verify that the results returned are grouped by the group by key flt_m

	for $x in dataset DsOne let $id := $x.id group by $m := $x.flt_m with $id return { "group-by-key": $m, "group-members": $id }	
9.	Test to group results based on an mandatory field of type point. for $x in dataset DsOne let $id := $x.id group by $m := $x.pt_m with $id return { "group-by-key": $m, "group-members": $id }	Verify that the results returned are grouped by the group by key pt_m
10.	Test to group results based on a mandatory field of type int64 for $x in dataset DsOne let $id := $x.id group by $m := $x.int_64_m with $id return { "group-by-key": $m, "group-members": $id }	Verify that the results returned are grouped by the group by key int_64_m
11.	Test to group results based on a mandatory field of type time for $x in dataset DsOne let $id := $x.id group by $m := $x.tm_m with $id return { "group-by-key": $m, "group-members": $id }	Verify that the results returned are grouped by the group by key tm_m
12.	Test to group results based on a multi-field group by key, which can be of the form [$x.int_m,$x.string_m] for $x in dataset DsOne let $id := $x.id group by $m := [$x.int_m,$x.string_m] with $id return { "group-by-key": $m, "group-members": $id }	Verify that the results returned are grouped by the multi-part group by key [$x.int_m,$x.string_m]

5.3.2. Tests to cover grouped aggregation.

The tests discussed in Table 7, can be slightly modified to iterate over the collection that holds the members of each group (i.e. the collection produced by the with clause for each of the groups) to perform aggregations over the data. All aggregate functions - min, max, sum, avg and count - will be tested for valid data that is part of the collection. These tests are discussed in Table 8. All tests will use the AllType schema and dataset definition DsOne as shown below,

```
create type AllType
as open {
    id : int32,
    int_m : int32,
    int_o : int32?,
    int_64_m : int64,
    string_m : string,
    string_o : string?,
    tm_m: time,
    dt_m: date,
    dt_tim_m: datetime,
    intrvl: interval,
    pt_m: point
  };

create dataset DsOne(TypeOpen) primary key id;
```

Table 8 : Grouped Aggregation Tests

	Test description & Aim	**Expected Results**
1	Test to group results based on a mandatory field of type int64 and perform aggregation over the collection produced by the *with* clause, this collection is an ordered list that has the members that belong to a group. Grouped aggregation : Get count of members of a group. for $x in dataset DsOne let $id := $x.id	Verify that the results are grouped and the grouped aggregation results are correct and verify from the query plan that we do a local pre clustered group by followed by a global pre clustered group by.

	group by $m := $x.int_64_m with $id return { "group-by-key": $m, "group-member-count": count($id) }	
2	Test to group results based on a mandatory field of type int64 and perform aggregation over the collection produced by the *with* clause, this collection is an ordered list that has the members that belong to a group. Grouped aggregation : Get the maximum number of members of a group. for $x in dataset DsOne let $id := $x.id group by $m := $x.int_64_m with $id return { "group-by-key": $m, "mx-group-member": max($id) }	Verify that the results are grouped and the grouped aggregation results are correct and verify from the query plan that we do a local pre clustered group by followed by a global pre clustered group by.
3	Test to group results based on a mandatory field of type int64 and perform aggregation over the collection produced by the *with* clause, this collection is an ordered list that has the members that belong to a group. Grouped aggregation : Get the average of members of a group. for $x in dataset DsOne let $id := $x.id group by $m := $x.int_64_m with $id return { "group-by-key": $m, "avg-group-member": avg($id) }	Verify that the results are grouped and the grouped aggregation results are correct and verify from the query plan that we do a local pre clustered group by followed by a global pre clustered group by.
4	Test to group results based on a mandatory field of type int64 and perform aggregation over the collection produced by the *with* clause, this collection is an ordered list that has the members that belong to a group. Grouped aggregation : Get the minimum of members of a group. for $x in dataset DsOne let $id := $x.id group by $m := $x.int_64_m with $id return { "group-by-key": $m, "min-group-member": min($id) }	Verify that the results are grouped and the grouped aggregation results are correct and verify from the query plan that we do a local pre clustered group by followed by a global pre clustered group by.

5.4 Order by clause

This section lists the tests that verify the correctness of the order by clause in AsterixDB. Order by functionality in AsterixDB was discussed in section 4.4 of this thesis. Data is sorted in ascending order by default in AsterixDB. The focus of these tests is to verify that the system orders data in the specified order when data is input to the order by clause from datasets that are hash partitioned across a cluster of nodes. Table 9 and Table 10 describe the order by tests that are executed on a standalone and cluster setup. The dataset used will be an internal dataset which is defined to be of an open type. Fields that are defined as optional will have some instances that will hold nulls. Results will be ordered by every field that is part of AllType schema, as a separate independent test, some of those tests are provided in Table 9. All tests in Table 9 will use the AllType schema and TestDs dataset definition as shown below,

```
create type AllType
as open {
    id : int32,
    int_m : int32,
    int_o : int32?,
    int_m : int64,
    int_m : int64?,
    string_m : string,
    string_o : string?,
    tm_m: time,
    dt_m: date,
    dt_tim_m: datetime,
    intrvl: interval,
    pt_m: point,
    flt_m: float
  };

create dataset TestDs(AllType) primary key id;
```

Table 9 : Tests to order results by primitive type attribute in order

	Test Description & Aim	**Expected Result**
1	Test to order results based on a mandatory field of type int32 for $l in dataset TestDs order by $l.int_m return $l	Verify that the result is in ascending order. Verify from the query plan that a local sort is applied in ascending order and the sorted data is then merged in ascending order.
2	Test to order results based on an optional field of type int32 for $l in dataset TestDs order by $l.int_o return $l	Verify that the result is in ascending order. Verify from the query plan that a local sort is applied in ascending order and the sorted data is then merged in ascending order.
3	Test to order results based on a mandatory field of type string for $l in dataset TestDs order by $l.string_m return $l	Verify that the result is in ascending order. Verify from the query plan that a local sort is applied in ascending order and the sorted data is then merged in ascending order.
4	Test to order results based on an optional field of type string for $l in dataset TestDs order by $l.string_o return $l	Verify that the result is in ascending order. Verify from the query plan that a local sort is applied in ascending order and the sorted data is then merged in ascending order.
5	Test to order results based on a mandatory field of type date for $l in dataset TestDs order by $l.dt_m return $l	Verify that the result is in ascending order. Verify from the query plan that a local sort is applied in ascending order and the sorted data is then merged in ascending order.

All tests in Table 10 will use the Emp schema and Employee dataset definition shown below:

```
create type Emp
as open {
  id : int32,
  name: string,
  name: string,
  salary: int32
}

create dataset Employee(Emp) primary key id;
```

Table 10: Tests to verify correctness of order by clause.

	Test Description & Aim	**Expected Result**
1.	Test to order the results (Records) returned by a select query based on primary key. The default order in which results are ordered in ascending order. for $l in dataset Employee order by $l.id return $l	Verify that results returned are in ascending order by default. Also verify from the query plan that there is no sort operation, since the ordering is based on the primary index.
2.	Test to order the results (Records) returned by a select query based on primary key in descending order. for $l in dataset Employee order by $l.id desc return $l	Verify that results returned should be in descending order. Also verify from the query plan that there is a sort operation that orders data in descending order.
3.	Test to order the results (Records) returned by a select query based on non key attribute. for $l in dataset Employee order by $l.name return $l	Verify that results returned are in ascending order by default. Verify from the query plan that a local sort is applied in ascending order and the sorted data is then merged in ascending order.
4.	Test to order the results (Records) returned by a select query based on non key attribute in descending order. for $l in dataset Employee order by $l.name desc return $l	Verify that results returned should be in descending order. Verify from the query plan that a local sort is applied in descending order and the sorted data is then merged in descending order.

5.	Test to order the results based on an undefined field of type string. Note that there is no field named “hobby” in the Emp type. for $l in dataset Employee order by $l.hobby desc return $l	Verify that results returned should be in descending order. Verify from the query plan that a local sort is applied in descending order and the sorted data is then merged in descending order.
6.	Test to order the results in descending order based on primary key and limit the results using limit clause. for $l in dataset Employee order by $l.id desc limit 10 return $l.salary	Verify that a local sort is applied on each of the nodes, and the specified limit is applied to the sorted data on each of the nodes and then the data is merged and the global limit is applied to the merged data. This will give the top N results.
7.	Use order by clause in a grouped aggregate query, to order the groups based on the grouping key. // order by the grouping key for $l in dataset Employee let $x := $l.id group by $m := $l.dname with $x order by $m return { "dname" : $m, "emp-count": count($x) }	Verify that results returned are in ascending order of dname field (order by the group by key).
8.	Test to use order by and limit clause and distinct clause in same AQL query. for $l in dataset Employee order by $l.id desc limit 2 distinct by $l.name return $l.salary	Verify that results returned are in descending order, and that there are no duplicates in the results and that the number of results returned are equal to the specified limit.
9.	Test to order results (employee name’s) and use limit clause. Get any five employees and sort the data by their names. for $l in dataset Employee limit 5 order by $l.name desc return $l.name	Verify that employee names are sorted in descending order. Verify from the query plan that we apply the global limit first and then do a sort operation. Note : This is because the limit appears before the order by clause in the AQL Query.

10.	Test to order results (employee name's) and apply limit to sorted data. Get the top 5 names of employees in descending order. for $l in dataset Employee order by $l.name desc limit 5 return $l.name	Verify that employee names are sorted in descending order. Verify from the query plan that we perform a sort on each node locally and then apply the local limit and merge the data and then apply the global limit over the merged data. Note : This is because the limit appears after the order by clause in the AQL Query.
11.	Test to order results returned by an equi-join. // ID is defined as a primary key in T1 and T2 for $m in dataset T1 for $n in dataset T2 where $m.ID = $n.ID order by $m.ID return { "m":$m,"n":$n }	Verify that join results returned are in ascending order. Verify from the query plan that a sort operation is performed after the join operation and that the sorted data is merged and projected.
12.	Test to order results based on multi-field order by key. for $l in dataset Employee order by $l.name,$l.salary desc return $l	Verify that the results are ordered by the attributes specified in the order by clause.
13.	Test to order any N results returned by a range search query. Note that in this test a secondary index is defined over the name field. for $l in dataset Employee where $l.name >= "A" and $l.name <= "Z" limit 5 order by $l.name desc return $l.name	Verify that the results are within the specified range and that any N of those results are sorted in descending order.

5.5 Limit clause

This section lists the different tests that will be executed to verify correctness of limit clauses in AsterixDB. Limits in AsterixDB were explained in section 4.5 of this thesis.

The focus of the tests will be to verify that limits are applied locally and a global limit is then correctly applied. Table 11 describes tests that were executed on both a standalone setup and a cluster environment. Tests in rows 1, 2 and 3 in Table 11 use the Emp schema and Employee dataset definition shown below,

```
create type Emp
as open {
  id : int32,
  name: string,
}

create dataset Employee(Emp) primary key id;
```

Table 11: Tests to verify correctness of limit clause

	Aim of test & test description	**Expected Result**
1.	Test to restrict results returned (using limit clause) by a select query without a predicate. for $l in dataset Employee order by $l.id desc limit 10 return $l	Verify from the query plan that there is a local limit and a global limit that is applied. Query should return only limit-number of results, in descending order.
2	Test to restrict results returned (using limit clause) by a select query without a predicate. for $l in dataset Employee order by $l.name desc limit 10 return $l	Verify from the query plan that data is sorted on each node in descending order and that a local limit is applied on the sorted data and a global limit is applied on the merged data. Query should return only limit-number of results, in descending order.
3	Test to limit the results and order results by a missing undefined field of an open type dataset. for $l in dataset Employee limit 5 order by $l.dname desc return $l.dname	Verify that results are returned in descending order and that only N results are returned as specified by the limit. Verify from the query plan that a limit is applied to the data on each node locally, and that the data is merged and a global limit is applied on the merged data and the results are then sorted in descending order.

4	Test to restrict results returned by a nested query.
	
	for $l in dataset Customers
	return {
	"cust":$l,
	"orders": for $m in dataset Orders
	where $l.id=$m.id
	limit 50
	return $m
	}
		Verify from the query plan that a limit is applied within the nested query. Nested query should return only limit-number of results.

5	Test to restrict results returned by an equi-join. for $m in dataset T1 for $n in dataset T2 where $m.ID = $n.ID limit 25 return { “m”:$m,”n”:$n }	Verify from the query plan that there is a local limit and a global limit that is applied. Query should return only limit-number of join results.

6	Test to apply limit to data that is inserted into the added open field “F” that can hold a value of any type in it.
	
	create type Emp
	as open {
	id: int32
	}
	
	create dataset Employee(Emp) primary key id;
	
	insert into dataset Employee({“id”:35, “F”:”John Doe”});
	insert into dataset Employee({“id”:15, “F”:”2013-06-01”});
	insert into dataset Employee({“id”:25, “F”:55});
	insert into dataset Employee({“id”:85, “F”:{{“scuba”,”diving”,”hiking”}}});
	insert into dataset Employee({“id”:39, “F”:3.14});
	
	for $l in dataset Employee
	limit 3
	return $l.F
		Verify that the results returned are as per the specified limit.

Chapter 6

Experiences and Interesting Scenarios

In this chapter we discuss a few of our experiences and some interesting scenarios that we encountered during execution of tests as part of this thesis work. While a few of the issues reported were trivial, some others were interesting. Some of the interesting issues were those that were observed when working with aggregations, limits, joins, and ordering.

Interesting issues reported while working with aggregations included aggregation queries initially failing to correctly roll up the answer to one final number. The count aggregation function failed to sum up the aggregated counts returned by different nodes because the return type of the sum aggregation function was restricted to return values up to a maximum threshold supported by the int32 type in AsterixDB. Another interesting issue noticed with the count aggregate function was that the count aggregate function returned a null as a final result if any of the elements of an ordered list was null, which is incorrect for the semantics of count in AQL. Also, the Count aggregate function returned a NullPointerException when it was given an ordered list of nested ordered lists or an un-ordered list of strings as input.

Restricting the number of results returned by a select query worked as designed on a single node, but to limit the results returned by a select query over a cluster of nodes, local limits were being performed as designed on each of the nodes, but the global limit operator failed to apply the global limit to the data received from all the nodes.

In the case of joins, the hybrid hash join method was not being selected for equi-joins of datasets R and S if the join criteria given was expression(R) = expression(S). Instead, a nested loops join was being selected. The AQL query that uncovered this issue was:

```
for $l in dataset DsOne
for $m in dataset DsTwo
where lowercase($l.name) = lowercase($m.name)
return { “name”: $m.name}
```

This was reported as a performance issue for join processing.

The indexed nested loops join was not selected if the entire key was not involved in the join criteria, whereas a prefix should have been sufficient to make the composite key useful. The test that uncovered this issue was, when a prefix of a two part primary key defined on (fname, lname) was used to probe the inner relation, the indexed nested loops join method was not chosen to perform the join. This was reported as another performance issue for join processing.

Indexed nested loops join hint was not picked up on when used in a join. For subquery predicates of the form, “where $R.attrA /* +indexnl */ >= $S.attrB”, and “where $R.attrA /* +indexnl */ <= $S.attrB” the system did not recognize these predicates and instead returned NullPointerExceptions.

In all, approximately 31 correctness and performance issues were identified and resolved within the scope of this case study. These included the examples discussed here as well as a number of other bugs and issues, including issues such as users being able to drop metadata datasets, poor handling of certain syntax errors, errors related to dropping and recreating the same user-defined function, ordering results based on a field that was part of a record which was in turn an element of an ordered

list of records, comparisons between incompatible types not resulting in an exception, and so on.

Chapter 7

Conclusion

The goal of this thesis was to increase confidence in the correctness of the parallel partitioned query processing features of the AsterixDB BDMS, and to uncover bugs, report them, and track them to closure, thereby improving both the quality and usability of the AsterixDB BDMS. We first reviewed the related work in the area of testing database management systems. We then gave an overview of AsterixDB and discussed the different parallel query processing abilities of the AsterixDB BDMS. We then explained how we verified the correctness of AsterixDB's parallel query processing abilities such are join methods, aggregates, grouping, ordering, and limits. We shared a few of our experiences and interesting scenarios from tests that were performed on those parallel query processing abilities of AsterixDB. The main focus was towards verifying the correctness of operations globally, over a cluster of nodes, for data that was partitioned across many nodes in the cluster. All issues (i.e. bug reports) were reported on AsterixDB's GoogleCode issues page whenever incorrect or unexpected behavior was observed after executing the tests. These issues were prioritized and, based on the severity and importance of the issue, the issue was fixed appropriately by the development team.

It will be important for future work to continue in the direction of testing the AsterixDB BDMS. One additional clause in AQL that would benefit from the same kind of partitioned parallelism testing is the 'distinct by' clause. This clause is closely related

to 'group by' and 'order by' in nature, so the same methods used here could be (and will be, next) applied to verify AsterixDB's parallel handling of queries that include this duplicate elimination clause in AQL. A framework to execute rigorous concurrency tests and multi-user tests to verify and validate the correctness of the new transaction management system in AsterixDB is greatly needed. Crash recovery is another important feature that needs to be more heavily tested to ensure that the system can recover from unexpected failures such as disk failures, node failures etc. Metadata caching also needs to be more fully tested by executing related DDL statements from different sessions. Datafeed testing is another area that needs more work, in the future, to cover the different types of data that AsterixDB can ingest continuously from different sources (e.g., Twitter). Last but not least, code coverage is another strategy that can and should be used to identify what parts of the code base have been executed, at a statement level and/or function level, and which part of the code is unexercised at present, by running tests against the code base.

References

[1] DeWitt, D. and Gray, J. 1992. Parallel database systems: The future of high performance database systems. *Commun. ACM* 35, 6

[2] Graefe, G., 2011. Modern B-Tree Techniques. *Foundations and Trends in Databases*

[3] Selinger, P. G. et al, 1979. Access path selection in a relational database management system. *Proc. ACM SIGMOD Int'l. Conf. on Management of Data*

[4] Shapiro, L. D. 1986. Join processing in database systems with large main memories. *ACM Trans. Database Systems*

[5] Hyracks: A Flexible and Extensible Foundation for Data-Intensive Computing

[6] ASTERIX: towards a scalable, semi-structured data platform for evolving-world models

[7] XQuery 1.0: An XML query language. http://www.w3.org/TR/xquery/

[8] Stonebraker, M., et al.: MapReduce and parallel DBMSs: friends or foes? Commun. ACM

[9] Ramakrishnan, R., Gehrke, J.: Database Management Systems. WCB/McGraw-Hill, Boston

[10] JSON. http://www.json.org/

[11] Jaql, http://www.jaql.org

[12] Hyracks project on Google code. http://code.google.com/p/hyracks

[13] Graefe, G.: Query evaluation techniques for large databases. ACM Computing Surv. 25(2), 73–170(1993)

[14] Google protocol buffers. http://code.google.com/apis/protocolbuffers/

[15] Facebook Thrift. http://incubator.apache.org/thrift

[16] Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. In: OSDI, pp. 137–150 (2004)

[17] Chaiken, R., Jenkins, B., Larson, P.-Å., Ramsey, B., Shakib, D., Weaver, S., Zhou, J.: SCOPE: easy and efficient parallel processing of massive data sets. PVLDB 1(2), 1265–1276 (2008)

[18] Carey, M.J., Muhanna, W.A.: The performance of multiversion concurrency control algorithms. ACM Trans. Comput. Syst. 4(4), 338–378 (1986)

[19] Borkar, V., Carey, M., Grover, R., Onose, N., Vernica, R.: Hyracks: a flexible and extensible foundation for data-intensive computing. In: ICDE (2011)

[20] Apache Avro, http://hadoop.apache.org/avro/

[21] Apache Hadoop, http://hadoop.apache.org

[22] Amer-Yahia, S., Botev, C., Buxton, S., Case, P., Doerre, J., Dyck, M., Holstege, M., Melton, J., Rys, M., Shanmugasundaram, J.: XQuery and XPath full text 1.0. W3C Candidate Recommendation, July 9 (2009)

[23] Olston, C., Reed, B., Srivastava, U., Kumar, R., Tomkins, A.: Pig Latin: a not-so-foreign language for data processing. In: SIGMOD Conference, pp. 1099–1110 (2008)

[24] Interpreting the Data: Parallel Analysis with Sawzall
Rob Pike, Sean Dorward, Robert Griesemer, Sean Quinlan

[25] Massive Stochastic Testing of SQL, Donald R Slutz, VLDB '98

[26] Kristi Morton , Nicolas Bruno, FlexMin: a flexible tool for automatic bug isolation in DBMS software, Proceedings of the Fourth International Workshop on Testing Database Systems, p.1-6, June 13-13, 2011, Athens, Greece

[27] Shadi Abdul Khalek , Sarfraz Khurshid, Automated SQL query generation for systematic testing of database engines, Proceedings of the IEEE/ACM international conference on Automated software engineering, September 20-24, 2010, Antwerp, Belgium

[28] Nicolas Bruno, Minimizing database repros using language grammars, Proceedings of the 13th International Conference on Extending Database Technology, March 22-26, 2010, Lausanne, Switzerland

[29] M. Muralikrishna, Using the optimizer to generate an effective regression suite: a first step, Proceedings of the Third International Workshop on Testing Database Systems, p.1-6, June 07-07, 2010, Indianapolis, Indiana

[30] Hicham G. Elmongui , Vivek Narasayya , Ravishankar Ramamurthy, A framework for testing query transformation rules, Proceedings of the 35th SIGMOD international conference on Management of data, June 29-July 02, 2009, Providence, Rhode Island, USA

[31] Florian Haftmann , Donald Kossmann , Eric Lo, Parallel execution of test runs for database application systems, Proceedings of the 31st international conference on Very large data bases, August 30-September 02, 2005, Trondheim, Norway

[32] Chaitanya Mishra , Nick Koudas , Calisto Zuzarte, Generating targeted queries for database testing, Proceedings of the 2008 ACM SIGMOD international conference on Management of data, June 09-12, 2008, Vancouver, Canada

[33] Eric Lo , Carsten Binnig , Donald Kossmann , M. Tamer Özsu , Wing-Kai Hon, A framework for testing DBMS features, The VLDB Journal — The International Journal on Very Large Data Bases, v.19 n.2, p.203-230, April 2010

[34] Testing the accuracy of query optimizers, Zhongxian Gu, Mohamed A. Soliman, Florian M. Waas; May 2012 DBTest '12: Proceedings of the Fifth International Workshop on Testing Database Systems

[35] QAGen: generating query-aware test databases, Carsten Binnig, Donald Kossmann, Eric Lo, M. Tamer Özsu; June 2007, SIGMOD '07: Proceedings of the 2007 ACM SIGMOD International conference on Management of data

[36] Parallel data generation for performance analysis of large, complex RDBMS Tilmann Rabl, Meikel Poess June 2011
DBTest '11: Proceedings of the Fourth International Workshop on Testing Database Systems

[37] Constraint-based test database generation for SQL queries
Claudio de la Riva, María José Suárez-Cabal, Javier Tuya
May 2010; AST '10: Proceedings of the 5th Workshop on Automation of Software Test

[38] Agrawal R, Carey MJ, Livny M (1987) Concurrency control performance modeling: alternatives and implications. ACM Trans Database Syst 12(4): 609–654

[39] Carey MJ, Livny M (1988) Distributed concurrency control performance: a study of algorithms, distribution and replication. In: Proc of the 14th Intl. Conf. on Very Large Database Systems, Los Angeles, California

[40] Carey MJ, Stonebraker M (1984) The performance of concurrency control algorithms for database management systems. In: Proc of the 10th Intl. Conf. on Very Large Database Systems, Singapore

[41] Testing challenges for extending SQL server's query processor: a case study
Torsten Grabs, Steve Herbert, Xin (Shin) Zhang
June 2008 DBTest '08: Proceedings of the 1st international workshop on Testing database systems

[42] Multiprocessor hash-based join algorithms
David J. DeWitt, Robert H. Gerber
VLDB '85 Proceedings of the 11th international conference on Very Large Data Bases

[43] The Gamma Database Machine Project (1990)
David J. Dewitt , Shahram Ghandeharizadeh , Donovan Schneider , Allan Bricker , Hui-i Hsiao , Rick Rasmussen

[44] DB2 Parallel Edition
C K Baru et al.

[45] Testing the Accuracy of Query Optimizers
Zhongxian Gu University of California Davis
Mohamed A. Soliman Greenplum/EMC
Florian M. Waas Greenplum/EMC
DBTest '12 Proceedings of the Fifth International Workshop on Testing Database Systems

[46] G. Graefe and K. Salem, editors. Proc. of the Fourth International Workshop on Testing Database Systems, DBTest 2011, Athens, Greece. ACM, 2011.

[47] J. Gray et al. “Quickly Generating Billion-Record Synthetic Databases,” Proceedings of the ACM International Conference on Management of Data (SIGMOD), 1994.

[48] A Parallel General-Purpose Synthetic Data Generator
Joseph E. Hoag, Craig W. Thompson
DBTest '11 Proceedings of the Fourth International Workshop on Testing Database Systems

[49] Symbolic execution and program testing
James C. King, IBM Thomas J. Watson Research Center, Yorktown Heights,

NY Communications of the ACM, Volume 19 Issue 7, July 1976

[50] Lohr, S The age of big data The New York Times February 12, 2012

[51] ASTERIX: scalable warehouse-style web data integration
Proceedings of the Ninth International Workshop on Information Integration on the Web Sattam, A et al.

[52] Big Data Platforms: What's Next ?
Vinayak Borkar, Michael J Carey and Chen Li

[53] Some experimental results on distributed join algorithms in a local network
Hongjun Lu, Michael J. Carey
VLDB '85 Proceedings of the 11th international conference on Very Large Data Bases

[54] Database System Concepts, Sixth Edition
Abraham Silberschatz, Henry F. Korth, S. Sudharshan

[55] Implementing Sorting in Database Systems
GOETZ GRAEFE, Microsoft
ACM Computing Surveys, Volume 38 Issue 3, 2006

[56] Nested Loops Revisited
David J. DeWitt, Jeffrey F. Naughton, Joseph Burger
PDIS '93 Proceedings of the 2nd International Conference on Parallel and Distributed Information Systems

[57] Revisiting Aggregation for Data Intensive Applications: A Performance Study.
Jian Wen, Vinayak Borkar, Michael Carey, Vassilis Tsotras
Note: This paper is under preparation

[58] Avrilia Floratou, Nikhil Teletia, David J. DeWitt, Jignesh M. Patel, and Donghui Zhang. 2012.
Can the elephants handle the NoSQL onslaught?.
Proc. VLDB Endow. 5, 12 (August 2012)

[59] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010.
Benchmarking cloud serving systems with YCSB.
In Proceedings of the 1st ACM Symposium on Cloud computing (SoCC '10).

[60] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. 2009.
A comparison of approaches to large-scale data analysis.
In Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data (SIGMOD '09).