UNIVERSITY OF CALIFORNIA,
IRVINE

External Data Access and Indexing in a Scalable Big Data Management System

THESIS

submitted in partial satisfaction of the requirements
for the degree of

MASTER OF SCIENCE

in Computer Science

by

Abdullah Alamoudi

Thesis Committee:
Professor Michael J. Carey, Chair
Professor Sharad Mehrotra
Professor Cristina Lopes

2014

© 2014 Abdullah Alamoudi

DEDICATION

To my father
Abdulrahman Alamoudi
and my mother
Lina Aldorobi
for their unconditional love and support.
To my brothers
Ahmed and Omar
and my sisters
Fatimah, Maryam, Sarah, Arwa, Dana and Reem.
To all those who struggle to make the world a better place.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like first to thank my exceptional advisor Prof. Michael Carey for believing in me and for giving me the chance to work with AsterixDB's team. I would like to thank him for all the knowledge he transferred to me and for his guidance and support throughout the research work. I would like to thank him for being an inspiration to all of those around him, for caring and for dedicating his time to help us all overcome our challenges. I consider myself very lucky to have had the chance to work with him. Nothing I say can express my gratitude and I will always be in his debt.

I would like to thank Vinayak Borkar for his sharp insights and valuable input in the design stage of this work. His ability to analyze and solve problems is extraordinary and his contribution to the whole AsterixDB project is invaluable. I would like to also thank Raman Grover for his excellent work. My thesis work was built on top of his work which I found to be of very high quality. I would like to thank him for his help with the writing of the thesis and for reviewing thousands of lines of code for me. I would like to also thank Zachary Heilbron for helping me understand the storage management design and code and for providing clear and complete answers to all of my endless questions and reviewing my code. I would like also to thank Young-Seok Kim for his help with understanding the transaction management part of AsterixDB and for reviewing the transaction part of the external indexing feature. I would like to also thank Sattam Alsubaie for providing moral support during hard times and for code reviewing the external indexing storage part of external indexing. I would like to thank everyone in AsterixDB's team for their contribution to this project and for creating a professional work environment.

I would like to thank my thesis committee members Prof. Sharad Mehrotra and Prof. Cristina Lopes for reviewing my thesis.

I would like to thank Saudi Aramco Management for providing this scholarship and for their continuous support throughout my professional career.

This thesis was supported in part by NSF grant CNS-1059436.

ABSTRACT OF THE DISSERTATION

External Data Access and Indexing in a Scalable
Big Data Management System

By

Abdullah Alamoudi

Master of Science in Computer Science

University of California, Irvine, 2014

Professor Michael J. Carey, Chair

Traditional Database Management Systems (DBMS) offer a long list of quality attributes such as high performance, a flexible query interface, accuracy, reliability and fault tolerance. However, in order for users to get these benefits, they need to first have their data loaded into the system and stored in its storage layer using the system's binary format and utilizing its different data structures. The space requirements and the computational and operational cost of loading data is unjustifiable at times. This cost increases as data becomes larger and larger, especially when existing systems are generating these data continually, e.g., by producing system logs. For these reasons, many existing applications don't use DBMSs at all and instead rely on custom scripts or specialized code that lack the qualities offered by DBMSs. This problem has been acknowledged by many Data Management Systems that also provide ways for users to use their query language to carry out different analysis against data in raw format. However, external data access in most of these systems involves expensive full scan operations, affecting the performance of these DBMSs to a great extent. For this reason, many data management systems provide external data access to facilitate data loading and not for ongoing use for routine data querying and analysis.

Recently, several research projects have sought to improve efficiency for external data access using different techniques. Each of these techniques has certain limitations, such as having to change the existing external data or having to write it in the first place through a specialized system, or has resulted in very small performance gains. In AsterixDB[2], the big data management system developed in UCI, we have designed and implemented a new feature that allows building incrementally refreshable indexes over external data. In this thesis, we explain in detail the different types of external data AsterixDB can access through its adapters. We then explain the semantics and user model associated with the indexing of external data. We follow that with a discussion of the system design for indexing external data and show how the system addresses the different challenges associated with this task. We further provide an evaluation of query performance over external data that resides in Hadoop Distributed File System. We compare AsterixDB's external data access with Hive on the same data files and with the same data after loading it into AsterixDB's internal storage. We show that a user can get competitive performance using AsterixDB without having to first load their data into the system.

Chapter 1

Introduction

Database Management Systems use many techniques discovered and prove to yield good performance results through decades of research and practice. These techniques cover different areas such as storage structures, access methods, caching, and cost efficient query compilation. A prerequisite for using these techniques is for the data management system to have total control over users' data, i.e., having it stored in the system's storage and modified through its interface. Loading data into the system can be a major obstacle in this era of big data, with data being produced in huge amounts by multiple systems, and being stored in different file systems using different formats.

Dealing with data that resides outside a data management system introduces a number of challenges. The challenges are direct implications of not having control over the binary representation of records, the storage location, and the data modification. To address these challenges, designers of an external data access facility need to decide which external resource types and which formats of records should their facility access. They also need to decide whether a user can extend their facility to support access of other types of data sources and formats and how to make that as painless as possible for the user. At the same time, the data management system must be equipped to deal with different sources of failures such as corrupted data or unreachable sources.

Providing basic full scan feature for accessing external data is a good step that enables users to carry out their analysis using a robust and well tested query facility rather than writing their own bug-prone analysis code. This does not, however, result in good performance and may leave the user frustrated by long query response times. We believe that the field of data management has matured to a point where it can provide better answers for users with such needs. Unfortunately, providing more complex techniques to speed up query processing increases the challenges and risks of dealing with external data.

In AsterixDB, we chose to add indexing capabilities for external data. Indexes are great for fast lookup queries and also allow the system to perform index nested-loop joins (which can be faster than other join algorithms for select/join cases). The decision to index external data carries with it another set of decisions. These decisions include, but are not limited to, which types of external data to index, which formats, the granularity of the indexes, whether to allow indexing of changing data or only static data, how to check for data changes outside the system, how to react to these changes, whether to allow more than one index per dataset, how to ensure consistency among different access methods, when and how to update indexes when data changes take place, what guarantees the system provides and how it maintains a consistent state in the face of different failures. Looking at these challenges, it can be clearly seen that this problem is not a trivial one and that the solution needs to be carefully designed and implemented in order to achieve its practical use.

In this thesis, we start by presenting the motivation behind this work as well as some background information about AsterixDB and the external dataset support that it had before adding the indexing feature. We then explain the semantics for indexing external data in AsterixDB, the syntax of the different commands associated with external indexes and how the system behaves in different states. We then explain how we chose to deal with each of the challenges listed above. We finish the thesis with an evaluation of external data access performance both with and without external indexes.

Chapter 2

Motivation

The performance of queries that access external data in traditional DBMSs has usually been worse than queries that access loaded records [12]. This gap in performance is not only attributed to having to scan all the records. In addition, the DBMS has to parse records and convert them to a format that can be understood by its internal operators. Moreover, in parallel DBMSs, load balancing is usually restricted by the location of data and its access APIs, which could potentially cause many nodes of the system to not participate in the reading and parsing steps. For these reasons, users with external data often resort to writing their own tools and scripts to perform their analyses. Writing optimized scripts requires experience in both scripts writing and query optimization techniques, i.e., pushing selection and projection down, perform the most selective filtering first,etc. Even for experienced users, this overhead is not a small one since writing a 1-2 line query using a declarative query language usually translates to tens or hundreds of lines in a scripting language [12]. Recently, the database management community has started considering different approaches to speeding up the performance of external data access, with each approach targeting one of the bottlenecks.

In the summer of 2013, a company with a very large Hadoop cluster was briefed on AsterixDB and was quite impressed with the design and performance of AsterixDB as compared to other big data management options. However, they didn't have enough free storage space to consider moving their data into AsterixDB's internal storage and proposed adding support for indexing of data that resides outside the system. The availability of efficient indexes enables data management systems to perform record lookup operations with very low cost regardless of the size of the accessed dataset, and indexes can also be used to speed up join operations; the company wanted these benefits for external data. Two weeks later, another collaborating company provided a similar request, which quickly reinforced our growing belief of a very real need for indexing of external data - especially for "Big Data".

Looking at the Hadoop user community, the need for indexing of Hadoop Distributed File System data is very clear. Analysts are often not allowed to run ad-hoc unplanned data analysis jobs due to the high cost of running these jobs. Instead, they have to plan and go through long workflows to incorporate their analysis into the system's planned Map and Reduce jobs. This can take days or even weeks especially if the analyst wants to query records that were generated over a long period of time. An analyst at this stage may have to choose between going through this workflow and waiting to get the result or relying on previously computed aggregates which don't provide accurate results. Indexing of HDFS data would provide a third attractive alternative that could allow analysts to run queries at any time with minimal cost.

Aside from the clear performance benefits, this work provides a complete user model with clearly defined semantics that may inform future systems that wish to adapt a similar approach towards supporting external data. In addition, indexing of external data provides an easy way for potential users to try and test the system's secondary index performance and other features of its query language and execution platform without having to preload their data to do so.

Chapter 3

Related Work

The need for accessing external data has been long recognized in the data management industry and was even added to the SQL standard as described in ”SQL and Management of External Data” [16]. Leading DBMSs provide raw file scanning capability via features such as external tables in Oracle and MySQL[4] and via the external link and Open Row Set features in MS SQL. Some also allow users to extend this feature. Oracle supports table functions, which are user defined functions that return sets of rows that are fed into query pipelines in a way similar to regular database tables. MS SQL provides a similar feature using Table-Valued User-Defined functions. These systems essentially perform full table scanning and parsing when answering a query that needs to access data residing outside the system. This means that queries that run on raw data files will likely perform worse than queries that access loaded data.

In the data management research community, different studies related to accessing external data were published. In a study that compares DBMSs query performance with the use of UNIX tools, a system called FlatSQL was designed and implemented to perform SQL queries over text files [15]. FlatSQL's performance was compared against UNIX

Grep and against a DBMS and was found to perform better in some special cases, but it performed much worse in the majority of cases. In the paper "Here are my Data Files. Here are my Queries. Where are my Results?" [12], an eloquent case is presented for the need to design and implement advanced techniques for dealing with data that resides outside a DBMS in order to accommodate the needs of the external data user base. This study lead to the creation of the NoDB research project [6]. NoDB employs caching of data, adaptive partial loading, building of positional map structures to perform selective parsing, slicing table attributes into different files, pushing projection into the I/O level, and reducing the needed computation by parsing only needed attributes.

The high cost of loading data into a parallel data management system in a big data warehouse is shown in another study [21]. In this study, the authors describe their approach to integrating Hadoop with Teradata EDW, dividing the work between two different systems. Another approach to reducing the time to first analysis was shown in Yale's HadoopDB project[5] in which users are allowed to run their queries directly on raw HDFS files using MapReduce operations reducing time to first analysis to zero while invisibly and incrementally loading data into the DBMSs storage layer in order to leverage the system's capabilities in future operations. To avoid performing full scan operations, a user driven tradeoff between accuracy and result quality can also be used to speed up query execution over text files[14].

Different approaches were tried to improve the performance of queries on HDFS data. Split-oriented indexes can be built over HDFS data to determine which splits need to participate in an analysis job and to reduce the number of mappers [10]. Another approach that aims at indexing Hadoop's data does that by building trojan indexes into the physical file splits when loading the data into HDFS storage using User Defined Functions (UDFs) [9]. During query processing, the trojan index part of each stored split is examined and the splits matching the search criteria are read.

AsterixDB's solution for boosting the performance of external data access is a unique solution in a number of ways. AsterixDB allows users to build distributed record-level indexes over external data. It doesn't cache or re-write data, the query results are always correct, and data is not loaded into the AsterixDB storage layer at any time. The presence of the indexes doesn't affect the external data itself, and AsterixDB supports indexing of multiple HDFS input formats. This is accomplished to a large extent by making externally stored data "look" like internally stored data from the perspective of the vast majority of the system's query processing components.

Chapter 4

Background

AsterixDB is a new BDMS (Big Data Management System) with a feature set that distinguishes it from other platforms in today's open source Big Data ecosystem. Its features make it well-suited to applications including web data warehousing, social data storage and analysis, and other use cases related to Big Data.

Development of AsterixDB began in 2009 and led to a mid-2013 initial open source release. AsterixDB runs on top of Hyracks, which is a partitioned-parallel software platform designed to run data-intensive computations on large clusters [7]. A complete description of the initial release of the system can be found in the paper ”AsterixDB: A Scalable, Open Source BDMS” [20]. In this chapter, we will very briefly describe AsterixDB's architecture, its data model, its query language and its storage and transaction model.

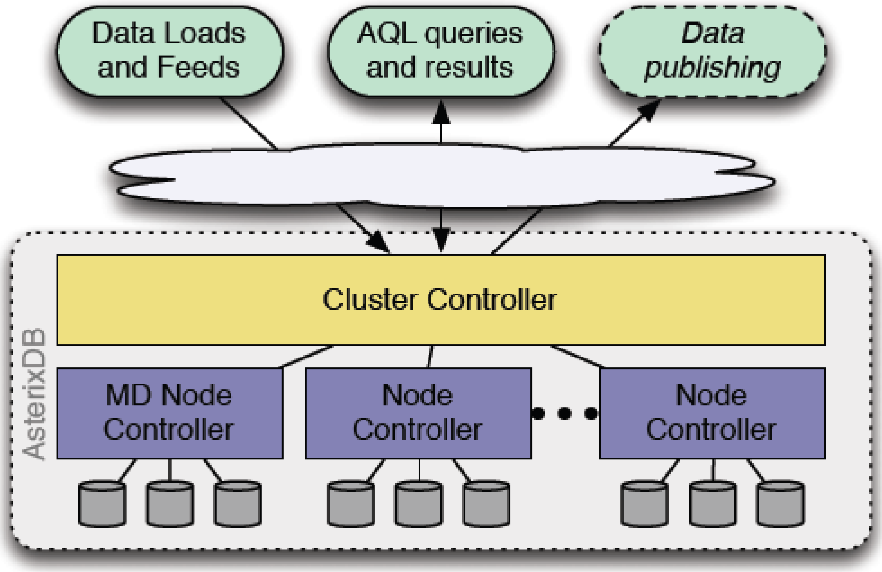

Figure 4.1: AsterixDB's architecture

4.1 AsterixDB Architecture

Figure 4.1 provides a high-level overview of AsterixDB and its basic logical architecture. Data enters the system through loading, continuous feeds, and/or insertion queries. Data is accessed via queries and the return (synchronously or asynchronously) of their results. AsterixDB aims to support a wide range of query types, including large queries (like current Big Data query platforms), short queries (like current key-value stores), as well as everything in between (like traditional parallel databases). The Cluster Controller in Figure 4.1 is the logical entry point for user requests; the Node Controllers and Metadata (MD) Node Controller provide access to AsterixDBs metadata and the aggregate processing power of the underlying shared-nothing cluster. The figures dotted Data publishing path indicates that we are also working towards the eventual inclusion of support for continuous queries. AsterixDB has a typical layered DBMS architecture that operates on nodes of shared nothing clusters. AsterixDB's query compiler compiles AQL queries into Hyracks jobs expressed as Directed Acyclic Graphs (DAGs) consisting

```
create dataverse tpch;
use dataverse tpch;
create type lineitemType as closed{
  lorderkey:int32,lpartkey:int32,
  lsuppkey:int32,llinenumber:int32,
  lquantity:double,lextendedprice:double,
  ldiscount:double,ltax:double,
  lreturnflag:string,llinestatus:string,
  lshipdate:date,lcommitdate:date,
  lreceiptdate:date,lshipinstruct:string,
  lshipmode:string,lcomment:string};
```

Listing 4.1: Creating a dataverse and a data type

of operators that perform different types of computations or I/O operations and connectors which connect operators and make the newly produced partitions available at the consuming operators' side. Hyracks jobs are executed in a pipelined fashion and data messages are passed between operators as frames containing sets of tuples.

4.2 AsterixDB Data Model

AsterixDB has its own data model called the AsterixDB Data Model(ADM). ADM is a superset of JSON, and each individual ADM data instance is optionally typed and self-describing. All data instances live in datasets which in turn live in dataverses that represent data universes in AsterixDB's world. Datasets may have associated schema information that describes the core content of their instances. AsterixDB's schemes are by default open, in the sense that individual data instances may contain more information than what their dataset schema indicates and can differ from one another regarding their extended content. Listing 4.1 shows an example of how one creates a dataverse and a data type in AsterixDB.

4.3 AsterixDB Query Language

Data in AsterixDB is accessed and manipulated through the use of the associated AsterixDB Query Language (AQL). AQL is designed to cleanly match and handle the data-structuring constructs of ADM. It is inspired by XQuery, but omits its many XML-specific and document-specific features. Listing 4.2 shows examples of AQL statements for creating, loading and querying an internal dataset.

```
use dataverse tpch;
create dataset InLineitem(lineitemType) primary key lorderkey,llinenumber
    ;
load dataset InLineitem using hdfs (("hdfs"="hdfs://sensorium-21.ics.uci.
    edu:54311"),("path"="/data/tpch/lineitem"),("input-format"="text-
    input-format"),("format"="delimited-text"),("delimiter"="|"));
for $l in dataset InLineitem
  where $l.lshipdate <= date("1998-09-02")
  order by $l.lextendedprice
return $l;
```

Listing 4.2: Creating loading and querying an internal dataset

4.4 AsterixDB Storage and Transaction Model

Indexes in AsterixDB are Log Structured Merge (LSM) indexes optimized for quick insert, update and delete operations, with every index consisting of several components stored in disk and a single memory component all sorted in a chronological order. During search operations, multiple cursors are used and all components of an index are searched, the results of the search are merged and returned. Deletion of tuples in LSM indexes is done through either physically deleting tuples found in the current memory component or else by inserting an "antimatter" tuple in the memory component (which indicates that the corresponding tuple is deleted) or by adding a "killer tuple" to a buddy b-tree that holds a "deleted list" of tuples. The transaction model supported by the indexes in AsterixDB is that of a typical NoSQL store: every record operation maintains ACID properties across the primary and secondary indexes of a dataset and is considered a single transaction by itself, so each record operation is either committed in all indexes of a dataset or else none of the indexes is affected. An example consequence of this is that, when an AQL statement attempts to insert 1000 records, it is possible that the first 800 records may end up being committed while the remaining 200 records fail to be inserted. More about AsterixDB's storage can be found in "Storage Management in AsterixDB"[19].

Chapter 5

External Datasets

In this chapter, we describe the state of the external data support in AsterixDB prior to the start of this thesis work and the initial improvements made by this author to the HDFS adapter.

AsterixDB allows users to define external datasets and use them to transparently query data stored in external sources using AQL. An external dataset in AsterixDB, like an internal dataset, is associated with a type, but doesn't have to have a primary key. In addition to the dataset's type, an external dataset has an associated adapter. Adapters in AsterixDB perform on-the-fly data fetching from an external source and do record parsing during query execution. AsterixDB is shipped with a set of built-in adapters, and users can also provide their own implementations that can be added to the system using AsterixDB's external library feature. A user implementing an adapter has the complete flexibility to manage any combination of resources and data formats. An adapter implementation also controls which nodes will participate in query execution. In addition to their use with external datasets, adapters are also used for loading records into an internal dataset and for the data ingestion tasks of data feed management [18].

AsterixDB does not cache any parsed data from external sources.

Creating an external dataset simply adds an entry for the dataset in the Metadata dataverse; in this case, the entry also holds the adapter type and adapter arguments associated with the dataset. Users are not allowed to carry out any operations that insert, delete or update external datasets. Deleting an external dataset only removes its entry from AsterixDB's metadata and doesn't affect the data at the source. Querying an external dataset in AQL is done using the same syntax used for querying an internal dataset as seen in Listing 5.1.

```
use dataverse tpch;
create external dataset ExLineitem(lineitemType) using hdfs (("hdfs"="
    hdfs://sensorium-21.ics.uci.edu:54311"),("path"="/data/tpch/lineitem"
    ),("input-format"="text-input-format"),("format"="delimited-text"), (
    "delimiter"="|"));
for $l in dataset ExLineitem
  where $l.lshipdate <= date("1998-09-02")
  order by $l.lextendedprice
return $l;
```

Listing 5.1: Creating and querying an external dataset

5.1 Query Planning and Execution

During query planning, the AQL compiler creates an external dataset scan operator for each external dataset in the user's query. This operator uses the dataset's adapter to parse records and package them into Hyracks frames sending them to the consuming operator in the query pipeline. When the compiler creates the adapter object, it uses the implementation for that adapter to determine which nodes to run the external scan operator on. For example, when the compiler creates the local file system adapter, the adapter will select the nodes which store files for the accessed external dataset to participate in the query execution and for HDFS adapter, all nodes are selected. During query execution, the adapter on each participating node first creates its appropriate parser and starts fetching the assigned part of the data from external sources, parsing the records and sending them to the next operator in the query pipeline.

5.2 Built-in Adapters

The built-in adapters in AsterixDB are of multiple types, each requiring different sets of arguments at creation time. Adapters use their arguments to locate and read data from different sources and to use an appropriate parser to parse records into AsterixDB's binary format for ADM data. The main resource types that can be accessed by built in adapters in AsterixDB are data files in the cluster nodes' Local File Systems, data files in HDFS, and data in web resources.

(1) **Data Files in Local File Systems**: The AsterixDB Local File System adapter is a built-in adapter for reading data from files residing in the local file system of AsterixDB's nodes. The arguments that this adapter looks for are a list of files and the data format. The format is used to determine the appropriate parser to be used for parsing that data. AsterixDB's built-in text parsers support parsing delimited text files, JSON, or ADM data format. When parsing self describing data such as data in JSON or ADM format, the user doesn't actually need to specify the schema of the dataset since it can be extracted from the data files and the parser can leverage AsterixDB's open fields capability to construct appropriate self-describing tuples for the dataset's records. The parallelism achieved by this adapter at runtime is determined by the number and

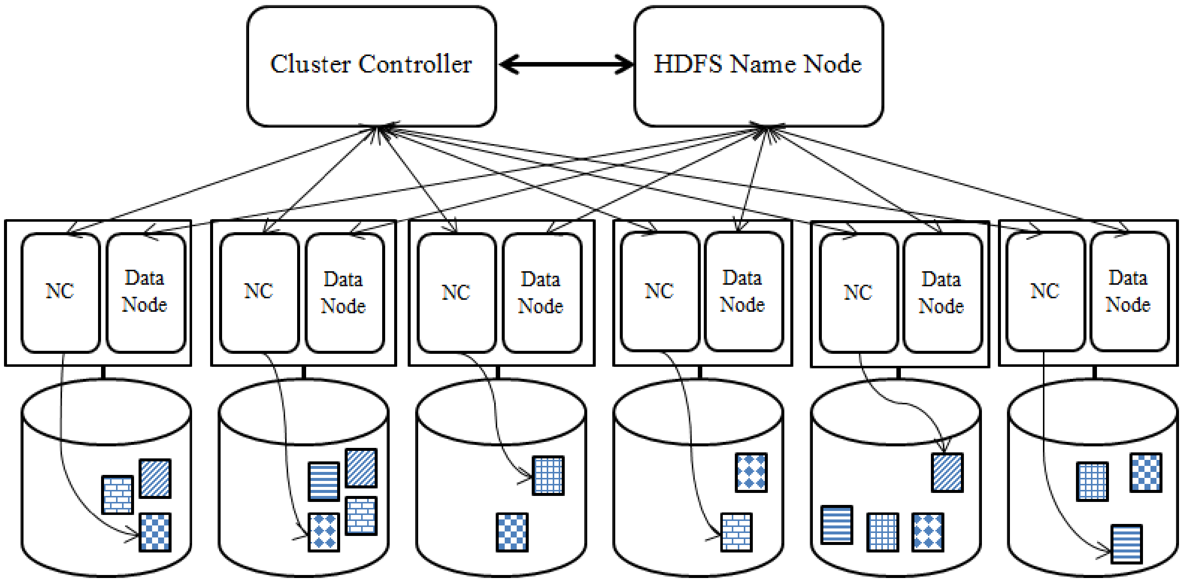

Figure 5.1: Cluster controller contacts HDFS's name node and schedules reading and parsing assignments

locations of the data files. Multiple files on the same node are read and parsed in parallel on that node, making use of multi-core processors in individual nodes, while having multiple files on different nodes increases cluster level parallelism since multiple nodes will be reading and parsing their local files at the same time.

(2) **Data Files in HDFS**: The AsterixDB HDFS adapter performs the same function for data files stored in HDFS, but with slight implementation differences. It expects an additional argument that describes the InputFileFormat of the dataset's files. When reading HDFS files, a file can be broken into logical byte splits that can be read concurrently by multiple readers running on different nodes. As shown in Figure 5.1, during the query planning phase AsterixDB's cluster controller contacts HDFS's name node to get the block storage information for files under that dataset. It then assigns the fetching and parsing responsibilities to those nodes giving higher priority to nodes that currently hold the corresponding HDFS Blocks in their local storage devices. In order to achieve higher level of parallelism, the scheduler may also choose to assign reading

and parsing of a file split to a node that doesn't have it locally in its storage device.

At the start of this project, the AsterixDB HDFS adapter was able to only access Text and Sequence Files and to parse only delimited text and ADM records. The first improvement made to the HDFS adapter was to enable it to access files of any of the HDFS input formats. The second improvements was adding a parser that uses Hive Serdes to read any binary format that can be deserialized using any of the Hive Serdes. This improvement has allowed AsterixDB to read many new binary formats, such as RCFiles, Avro data, Optimized RC, Regex, Thrift Byte Streams, and many more. The built-in Hive object parser performs 2-step parsing by first deserializing binary records into Hive objects using Hive's own code and then converting the produced objects into AsterixDB's binary format. The last improvement which provides greater flexibility, was to allow users to provide and use their own implementation of HDFS records parsers when the provided "out of the box" parsers don't meet their needs. All of these were possible before through implementing a complete user-defined adapter, but now require much less effort due to these improvements.

(3) **Data in web resources**: AsterixDB also offers a variety of adapters for reading data from web resources. When a dataset's records are read from the web, a set of URIs are used to identify the resources to be requested. Data is parsed using the same parsers available for local files' textual data formats, and the level of parallelism is determined by the number of URIs. Readers for the URIs are assigned by AsterixDB's compiler to different cluster nodes during query compilation time. At runtime, each participating node uses the assigned URIs to fetch and parse records into the ADM binary format.

Chapter 6

Indexing of External Data

Indexes are used heavily by data management systems to provide fast and efficient access to selected records that satisfy search predicates. Using indexes greatly reduces the cost of I/O and results in improved overall performance of the DBMS. Unfortunately, as mentioned earlier, traditional DBMSs do not support building indexes over external data in their external raw format, and thus require data to be loaded first into the system.

Building an efficient and robust indexing scheme over external data presents several challenges. When building an index, a DBMS needs to have a unique identifier for each indexed record in order for indexes to be able to ”point” to it. In addition, the DBMS must be able to use this record id (RID) to efficiently locate and fetch the record itself. In AsterixDB, an internal dataset record is identified by its unique primary key, which can be used to lookup the record in the dataset's primary index. External data comes in different container formats and different content formats, hence, appropriate RIDs must be identified for each targeted combination, extracted when building the index ,and used to access records when the index is selected as an access method. In a parallel DBMS, minimizing nodes' communication and choosing a good index distribution strategy is another challenge. A data management system which provides indexing of data must also provide clear and reasonable semantics and a well-defined user model in response to data changes. This includes making sure that queries will provide the exact same results when data is accessed through indexes, or through a full scan; the DBMS must also provide a mechanism for incrementally updating the indexes of an external dataset.

In this work, we have added an external dataset indexing feature to AsterixDB, allowing users to build distributed B-tree and R-tree indexes . These indexes can then be used to efficiently access records, reducing query response times greatly without first having to load the records into the system's storage at all. In this chapter, we first present the user experience and the access semantics and requirements for indexing an external dataset. We then, describe in detail, how we have implemented this feature and discuss the design decisions that were made in the process.

6.1 Semantics and User Experience

When an un-indexed external dataset is scanned, the scan operator reads and parses records found in the source locations that were defined when the dataset was created. When data changes take place in the source, the changes are seen by the next scan query that accesses the data. When a user creates an index or multiple indexes over a *static* external dataset, the indexes will always be up to date in terms of matching the state of the data at the source. Accessing a dataset in that case, using either an index or a full scan operator, produces the same output. In practice, however, most data changes overtime, with records being added, deleted, and/or updated, rather than being static. Having outdated indexes could then result in producing inconsistent results when a dataset is accessed using an index or using a scan operator. Moreover, creating indexes at different times might produce multiple indexes pointing to different data. To take care of these anomalies and provide well-defined behavior, the concept of an external dataset *snapshot* was introduced in AsterixDB. This concept facilitates understanding of data access semantics and clarifies the system's behavior in all scenarios. It is also used to preserve consistency among an external datasets' indexes in addition to enforcing a shared view of data between the index access operator and the dataset scan operator.

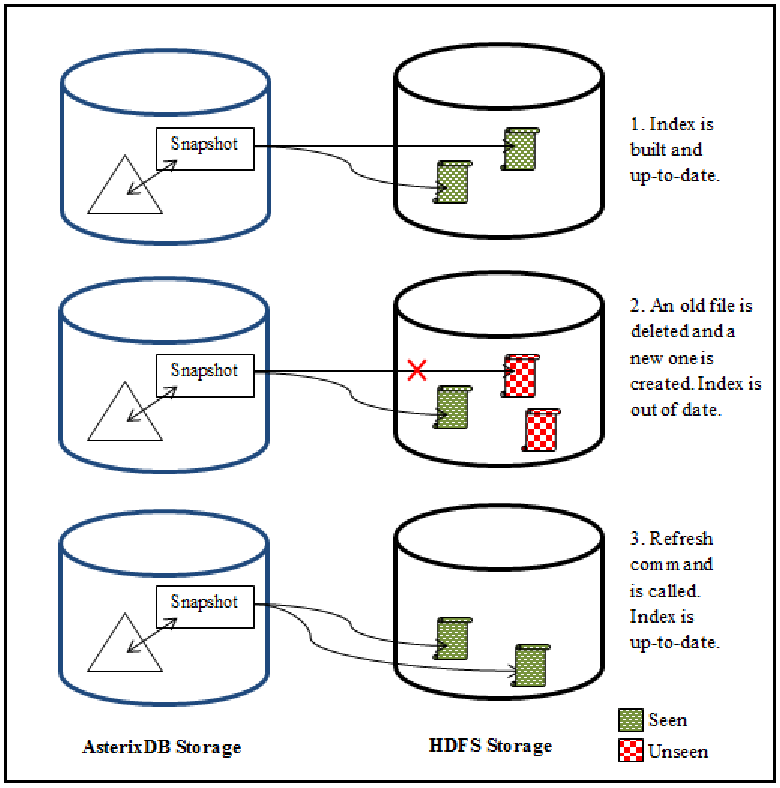
Snapshot
1. Index is built and up-to-date.
Snapshot
2. An old file is deleted and a new one is created. Index is out of date.
Snapshot
3. Refresh command is called. Index is up-to-date.
AsterixDB Storage
HDFS Storage
🟩 Seen
🟥 Unseen

Figure 6.1: Stored snapshot and file system as filters

An external data snapshot represents the state of an external dataset at a point in time. It consists of a list of file records. Each external file record contains the dataverse name and the dataset name of its external dataset. In addition, it stores the file's absolute path, modification time and its size at the snapshot capture time. The ExternalFile Metadata dataset in AsterixDB stores the snapshots of all the external datasets in the system and can be queried as illustrated in Listing 6.2.

When the first index over an external dataset is created, a dataset's snapshot is captured and the index is created according to the dataset's status at that point in time. When a user creates more indexes, they are created according to the captured snapshot of that dataset. When the system accesses an external dataset that has indexes, it only accesses records that were present as of the snapshot capture time. This means that an external dataset access will only read the intersection of (i) records present at the snapshot capture time and (ii) the existing records at the dataset's access time as shown in Figure 6.1. To update a dataset's snapshot and all of its indexes to the current point in time, a user can use the refresh external dataset statement. Queries are not interrupted by the refresh operation and to establish a defined behavior, queries that accesses the dataset must access the same version of the dataset index on all nodes. This means that refreshes and queries are logically serialized with respect to each other. In case of a failure during index creation, the index is dropped. The refresh statement on the other hand must be run as an atomic operation and in case of any failure, the dataset's snapshot and all its indexes must be restored to its previous state. In case of any failure during index creation, the index is dropped. The refresh statement also must be run as an atomic operation and in case of any failure, the dataset's snapshot and all its indexes must be restored to its previous state. To understand how important this transactional behavior, consider the case of an indexed external dataset that has tera bytes of records with multiple indexes that are refreshed daily. If after a long time of maintaining these indexes, a system crash took place and the system was not able to restore these indexes to a consistent state, a very costly and undesirable re-indexing of the whole dataset is required. Adding to that how common failures are when dealing with external systems stresses the importance of this property. The user can also query the stored snapshot of an external dataset that is kept in the Metadata ExternalFile dataset.

```
use dataverse tpch;
create index OrderKeyIdx on InLineitem(lorderkey);
create index OrderKeyIdx on ExLineitem(lorderkey);
```

Listing 6.1: Creating a B-Tree index over internal and external datasets

```
use dataverse tpch;
refresh external dataset  ExLineitem;
for $file in dataset Metadata.ExternalFile
  where $file.DataverseName = "tpch"
  and   $file.DatasetName = "ExLineitem"
return {
  "FileName":$file.FileName,
  "ModTime":$file.FileModTime,
  "Size":$file.FileSize
  };
```

Listing 6.2: Refreshing an external dataset and querying its stored files list

One of the desired qualities is to create a similar experience for external data users as for internal data users. For this reason, the build index statement over an external dataset has exactly the same syntax as for internal datasets. Listing 6.1 shows an example of the create index statement for both internal and external datasets. An example of the refresh dataset statement is shown in listing 6.2 a long with a query that returns the stored snapshot of the dataset after the refresh and example result of the query can be seen in Listing 6.3

6.2 Design and Implementation

In this section we discuss the design decisions that we made to address each of the challenges associated with indexing external data in a parallel data management system. We also describe some of the key implementation details of this work.

This section starts by discussing the selected types of external dataset for indexing. It then describes how indexes are partitioned and the rationale behind the chosen parti-

```
{ "FileName": "/data/tpch/lineitem/lineitem.tbl.1", "ModTime": datetime("
   2014-05-05T09:57:32.244Z"), "Size": 19959828307i64 }
{ "FileName": "/data/tpch/lineitem/lineitem.tbl.2", "ModTime": datetime("
   2014-05-05T09:58:28.441Z"), "Size": 20072834877i64 }
{ "FileName": "/data/tpch/lineitem/lineitem.tbl.3", "ModTime": datetime("
   2014-05-05T10:02:24.344Z"), "Size": 20073143660i64 }
```

Listing 6.3: An example ADM result of querying the ExternalFile dataset

tioning strategy. This is followed by a discussion of index size optimizations and the approach tp storage of a dataset's snapshot. After that, a detailed description of the index building and index access pipelines in Hyracks is presented. We describe, then, the design of the atomic dataset refresh operation. Finally, we show how the system maintains consistency in different cases and how different operations on external dataset run concurrently.

6.2.1 Indexable External Datasets

External datasets that use the AsterixDB HDFS adapter have many characteristics that make them good first candidates for indexing. These characteristics include their popularity in the big data community. On top of that, most HDFS files conform to a known set of input formats that specify how their records are stored physically and how they can be read regardless of their content format. Hence, records in files that share an input format can be identified using similar record ids. In addition, files in HDFS are immutable, meaning that once a record is written to a file, it can't be modified and it remains there until the file is deleted. Moreover, HDFS files are usually accessed in big data analysis systems using full scan operations, making it extremely expensive to run unscheduled ad-hoc analysis jobs; the needs for indexing such data is very clear. For these reasons, we chose to focus on indexing external data that resides in HDFS. The majority of the design can also be used with external data residing the cluster nodes' file systems.

HDFS is the file system beneath Hadoop[1] which itself is an open source implementation of Google's MapReduce project [8]. HDFS consists of a name node, which stores files' metadata information, and data nodes, which bear the responsibility of storing the data itself. Files in HDFS are partitioned into blocks, allowing parallel reads, and each block is replicated according to a replication factor to strengthen the system against hardware failures. AsterixDB supports building indexes over data that is stored in three of the input formats of HDFS: TextInputFormat, SequenceFileInputFormat, and RCInputFormat.

TextInputFormat is the HDFS format for plain text files with records separated by new line characters. A record within a file of this format can be identified using its byte offset. SequenceFileInputFormat is a commonly used HDFS format for storing key/value pairs. It is extensively used in MapReduce jobs where temporary outputs of Map operations are stored as Sequence Files. Sequence files can be uncompressed, record compressed or block compressed. Figure 6.2 shows the structure of an uncompressed or value compressed sequence file. For these first two, records within a single file, similar to records in TextInputFormat files, can be identified using their byte offset. For a block compressed sequence file, records are compressed and stored in groups. Each record group is read together in a single read call. For this reason, an additional field is needed to store the record order within its record group. In AsterixDB, we don't support indexing of data that is stored in block compressed sequence files.

RCFileInputFormat is another widely used input format for storing (relational) data in HDFS [11]. RCFiles provide efficient compression and support efficient attribute projection due to their columnar order. RCFiles are built on top of Sequence files with records being separated into row groups. Each row group is stored in a columnar order

```
– Header
– Record
  – Record length
  – Key length
  – Key
  – (Compressed?) Value
– A sync–marker
```

Figure 6.2: Structure of Sequence files

within the value part with its metadata being stored in the key part. As shown in Figure 6.3 and Figure 6.4, a sync marker is written once for every record group and the records in a group are read together. A record within a file of this format is identified using the byte offset of its row group in addition to its order within the row group. The nature of RCFile input format forces an index-based data access to read a complete row group if it contains any number of records matching the search criteria. Hence, the size of row groups and the ratio of the number of records of interest to the number of read row groups will each play an important role in determining the efficiency of index access for records stored using RCFile format.

In addition to identifying records within files, the files themselves must be identified in every RID since a dataset in AsterixDB could map to multiple physical external HDFS files. An individual external dataset file in AsterixDB is therefore identified using a combination of its **path** and its **modification timestamp**. RIDs of records in Text and Sequence files consist of (***file id***, ***record offset***), and RIDs of records in RCFiles consist of (***file id***, ***row group offset***, ***row order***).

```
– Header
– Record Group
– Key part
  – Record group length
  – Key length
  – Number_of_rows_in_this_records_group
  – Column_1_ondisk_length
  – Column_1_row_1_value_plain_length
  – Column_1_row_2_value_plain_length
  ...
  – Column_2_ondisk_length
  – Column_2_row_1_value_plain_length
  – Column_2_row_2_value_plain_length
  ...
– Value part
  – Compressed or plain data of [column_1_row_1_value,
  column_1_row_2_value ,....]
  – Compressed or plain data of [column_2_row_1_value,
  column_2_row_2_value ,....]
– Sync–marker
```

Figure 6.3: Structure of RC files

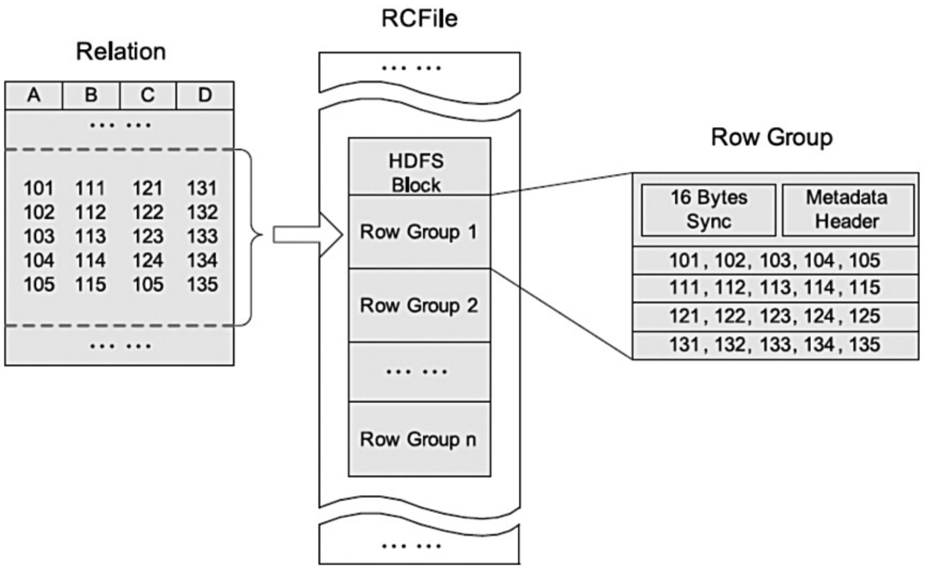

Figure 6.4: Logical and physical layouts of RC files

6.2.2 Index Partitioning

Partitioning of data is critical in any parallel system. It affects load balancing, scalability, and the level of needed communication between the system's nodes. Records of an internal dataset in AsterixDB are stored in primary B-tree indexes that are hash-partitioned on the primary key's values. This achieves an even distribution of tuples and allows partitioning the probing tuples when performing index nested-loop joins on the primary keys of a dataset. The secondary indexes of an internal dataset are co-located with the primary one on the same node, ensuring zero communication between nodes when secondary index access is used in a query at runtime.

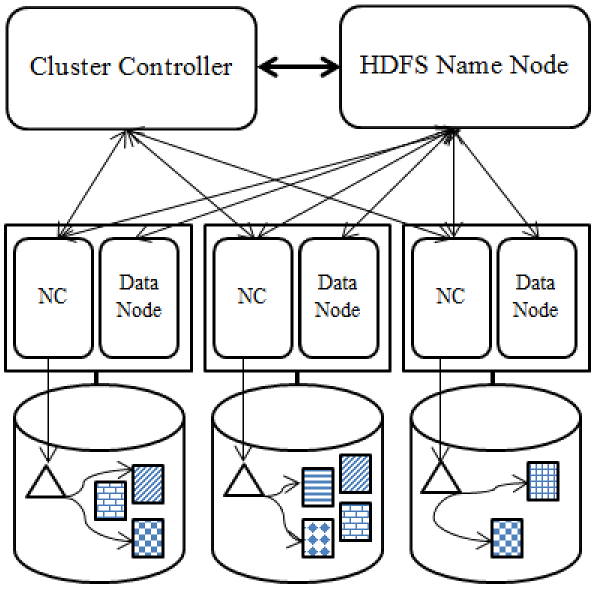

Figure 6.5: External dataset's indexes are co-located with HDFS blocks

Records distribution for external datasets is controlled by HDFS. The particular cluster nodes that store blocks that contain the records of a dataset can be either shared with AsterixDB nodes or separate but reachable from AsterixDB nodes. Moreover, the primary key of an external dataset's record, represented by its RID is not part of the dataset's data type and will never become a join attribute in any user query. In the case when the nodes of AsterixDB are shared with HDFS, external records will be indexed at one of the nodes that hold the containing HDFS data block, as can be seen in Figure 6.5. When choosing indexing assignments, AsterixDB keeps track of the number of HDFS blocks to index that have been assigned to each of its nodes. Since a block is replicated in more than one node, AsterixDB first determines the shared nodes among these and then sets the block's index destination to the node among them with the lower number of block assignments. (If indexed blocks are moved later by HDFS, they will still be accessible through the indexes, but the performance of the index access will be affected due to reduced data locality). In the case where cluster nodes are not shared between the two systems, block indexing responsibilities are assigned to AsterixDB nodes in a round-robin fashion to ensure an even distribution between all nodes.

6.2.3 The Files Index

Having the file name and modification timestamp appear in every RID in secondary indexes would require a relatively large amount of space. This would affect the dataset indexing and index access performance in various ways. The external sort operator in the index-building pipeline (seen in Figure 6.6) would be affected since the number of tuples that would fit in a frame would become smaller. The disk space required for storing the indexes would increase, affecting overall index performance. The sort operator in the index-access pipeline in Figure 6.8 would suffer as well since it sorts tuples on their RIDs. To mitigate these effects, we chose to replace the file id part of every RID with monotonically increasing integer values that are assigned to external files. AsterixDB stores each file's information in a B-tree index on disk, keyed by the assigned integer values, and uses this files index to fetch the needed file information at runtime.

The file information along with the assigned numbers are stored in the metadata ExternalFile dataset. Metadata datasets in AsterixDB are stored in a single special node called the metadata node. Using the ExternalFile dataset for file lookup during index access would force the system to either broadcast this information to different nodes or to have nodes request file information from the metadata node whenever they need to get a given file's information. This overhead could be negligible if the number of files is very small, but this is not always the case. For example, a dataset on HDFS with a retention period of 5 years and a 1 hour clock frequency would have 43800 files (assuming that each instance is stored in a single file). In such a case, broadcasting the file index would incur an unneeded cost, especially for queries that needs to access records in just a few files. At the same time, choosing to access the ExternalFile index when a node needs to access a new file could create a major source of communication overhead and may create a bottleneck in the metadata node. To avoid this, a files index is created for each indexed external dataset and is replicated in all AsterixDB nodes. This index is accessed in the lookup operator whenever it encounters a new file number and needs to get its path and modification time.

6.2.4 Hyracks Pipelines

AsterixDB uses Hyracks[7] as its execution engine, and all of its operations are expressed as Hyracks jobs. In this section, we describe the different operators involved in the index building and index access jobs.

When a user creates the first secondary index over an external dataset, AsterixDB communicates with the HDFS name node, gets a logical snapshot of the dataset, and stores it in its Metadata ExternalFile dataset. This snapshot is then replicated in all nodes of AsterixDB for use at index access time. A snapshot of an external HDFS dataset consists of a list of its files' metadata; each file's metadata record contains the file's absolute path, its modification time, and its current size in bytes. The path of the file along with its modification time identifies the file, while the size identifies the portion of the file available as of the snapshot capture time (The size is needed for maintaining a consistent view over each file since it turns out that HDFS might change the size during a write operation while keeping the same modification time). A file that has the same path with a different modification time is considered a new file. At the same time, records that are located beyond the captured size of the file are ignored, ensuring consistency when building different indexes and when the dataset is accessed using the scan operator.

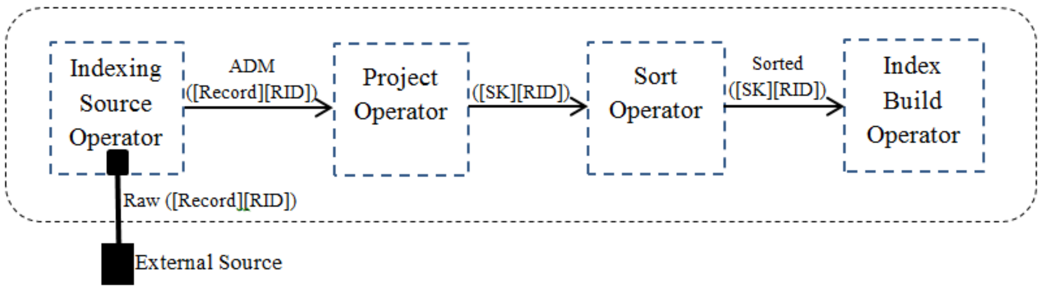

Figure 6.6: Hyracks pipeline for index building job

The Hyracks pipeline for index building consists of the four operators shown in Figure 6.6. Once the job is constructed, this pipeline becomes a completely local task in each data storage node. It starts with an external indexing source operator. When creating this operator, the cluster controller contacts HDFS and uses the stored snapshot of the dataset to create logical block-sized file splits using the intersection of the snapshot and the existing files in the file system. The splits are then assigned to nodes for the indexing task according to the partitioning strategy described above. During execution, the operator fetches and parses records, constructs their RIDs, and pushes them to the project operator as illustrated in Figure 6.7.

The project operator in Figure 6.6 extracts the secondary keys and the assigned RIDs

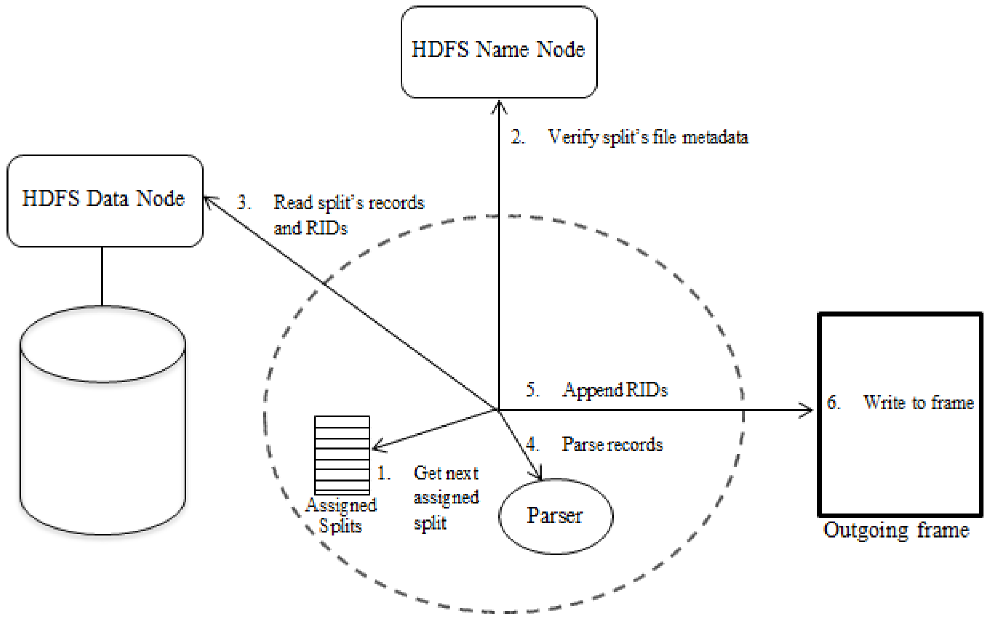

Figure 6.7: Hyracks operator for reading records with their IDs

and sends them to the sort operator which sorts them on the secondary keys. Finally, the index build operator builds the index itself bottom up.

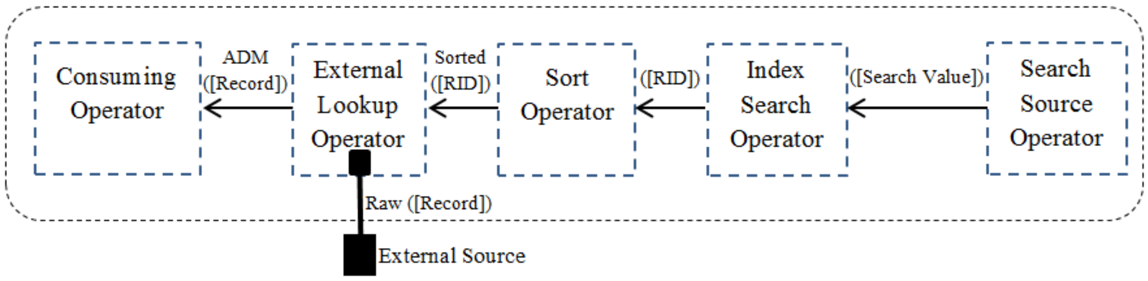

Figure 6.8: Hyracks pipeline for index access within a query

The pipeline for index access, shown in Figure 6.8 is completely local to each AsterixDB node as well. It starts with a source operator that supplies the search predicates followed by a secondary index search operator. This operator uses the incoming values to search the index and produce a list of RIDs pointing to records that match the search criteria. An external sort operator sorts the RIDs before feeding them to an external lookup operator. Having the RIDs sorted before access ensures that each external file is opened

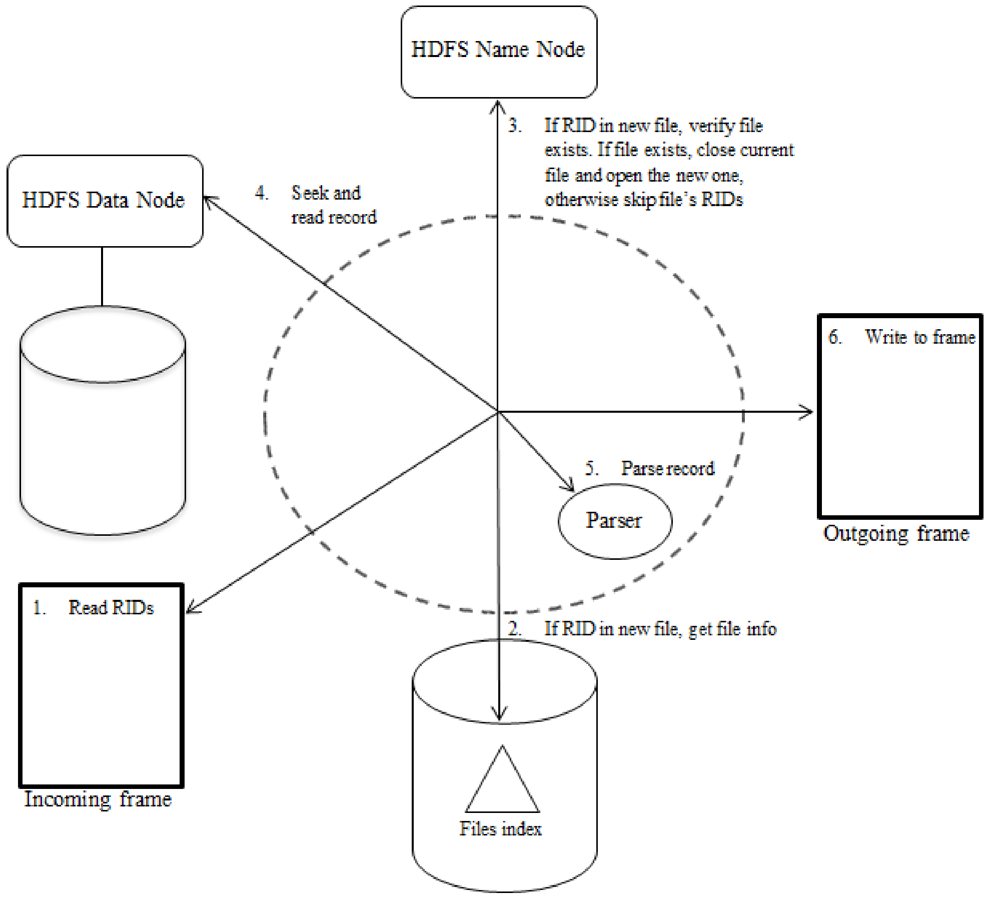
HDFS Name Node
3. If RID in new file, verify file exists. If file exists, close current file and open the new one, otherwise skip file's RIDs
HDFS Data Node
4. Seek and read record
6. Write to frame
Outgoing frame
5. Parse record
Parser
1. Read RIDs
Incoming frame
2. If RID in new file, get file info
Files index

Figure 6.9: Hyracks operator for external lookup in a query

once, and it also reduces I/O cost by reading records sequentially as they appear in HDFS blocks. The external lookup operator, illustrated in Figure 6.9, uses the incoming RIDs to reads records from external files. It monitors the file id field for file changes, and when the file id changes, it use the local replicated files index to get the stored file status as of the snapshot's capture time and contacts the HDFS name node to validate the continued existence of the file. If the file still exists, the operator opens the file and uses the incoming RIDs to fetch and parse its records. If the file was not found, then all incoming RIDs belonging to that deleted file are ignored. The records that are found are then parsed using the appropriate parser and pushed to the consuming operator in the pipeline. Figure 6.9 shows an illustration of how this operator works.

6.2.5 Atomic External Dataset Refresh

The data lifecycle in HDFS consists of two main operations, the addition and deletion of files. Different lifecycle models exist for different users' needs. The synchronous clocked dataset lifecycle is widely adopted in the Hadoop's users community. In this model, data has a retention period which represents a sliding time window over the data, with files being added and deleted according to a known frequency. With every addition of a new file of data, another file - older by the length of the retention period - is deleted. Usually, instances are identified by their creation time, which constitutes a part of the file URI. Oozie is a well known dataflow management and coordination system that works well with the synchronized synchronous dataset lifecycle, and other workflows have been designed for different data lifecycles [13]. Different systems write their outputs to HDFS files in different ways. Some systems first write their records to a temporary file and then rename it when all the records are written, while others write data directly to the destination file. Some systems add a single file per clock event, while others add a directory of files, while still other systems don't follow well-defined lifecycles. Due to these variations, we decided not to try to automate the refresh operation and we instead delegate this responsibility to the user of the system. At the same time, the AsterixDB manual refresh operation has a well-defined behavior that works with all of these use cases.

The external dataset refresh operation in AsterixDB advances the snapshot of the dataset to the current point in time, and it updates all of its secondary indexes to reflect the changes in HDFS that were captured by the new snapshot. In a cluster, AsterixDB follows The presumed-abort 2 Phase Commit (2PC) protocol[17] to perform the refresh as an atomic operation. The refresh operation starts by getting list of files statuses from HDFS's name node and comparing it with the list of files in the stored snapshot in AsterixDB, thus computing the snapshot delta. The computed delta consists of 3 sets: a set of deleted files, a set of record-appended files, and a set of new files. If the computed delta is empty, the operation is completed, and otherwise the delta is recorded on disk and the refresh transaction enters the (running) state. AsterixDB then adds the file delta to the replicated files index in all nodes and associates it with the refresh transaction. Locally, for each of the files indexes, a new hidden index component is created that holds all of the the refresh transaction's operations. For new files, new tuples are created in the transaction component, while for deleted files, anti-matter tuples are inserted into the same component. Transaction components can later be committed during the commit phase of the transaction and can also be deleted if the transaction aborts. After that, for each secondary index of the dataset, a Hyracks job is constructed to update the index under the ongoing transaction. This job consists of a pipeline similar to the index building pipeline. The indexing targets come from the delta rather than from the stored snapshot and the records are inserted into separate hidden index components. In addition to the insertion of new index entries, the bulk load operator in this pipeline performs deletions for records that belong to deleted files in older index components. Note that with only the file information at hand, deleting these entries in an index by adding anti-matter tuples would require an expensive full scan operation of the index. To avoid this, we added a buddy B-tree to each external index component. Each buddy B-tree contains file numbers of the deleted external files in previous components and it is filled in the bulk load operator of the refresh pipeline. The buddy B-tree then works as a filter during the search, as depicted in Figure 6.10.

During an index search, when the index search cursor finds a candidate tuple and before it returns it, it extracts the tuple's file number and uses it to probe all newer components' buddy B-trees. If the file number is found, the tuple is dropped, and otherwise the tuple is indeed returned by the cursor. Each buddy B-tree is accompanied by a Bloom filter

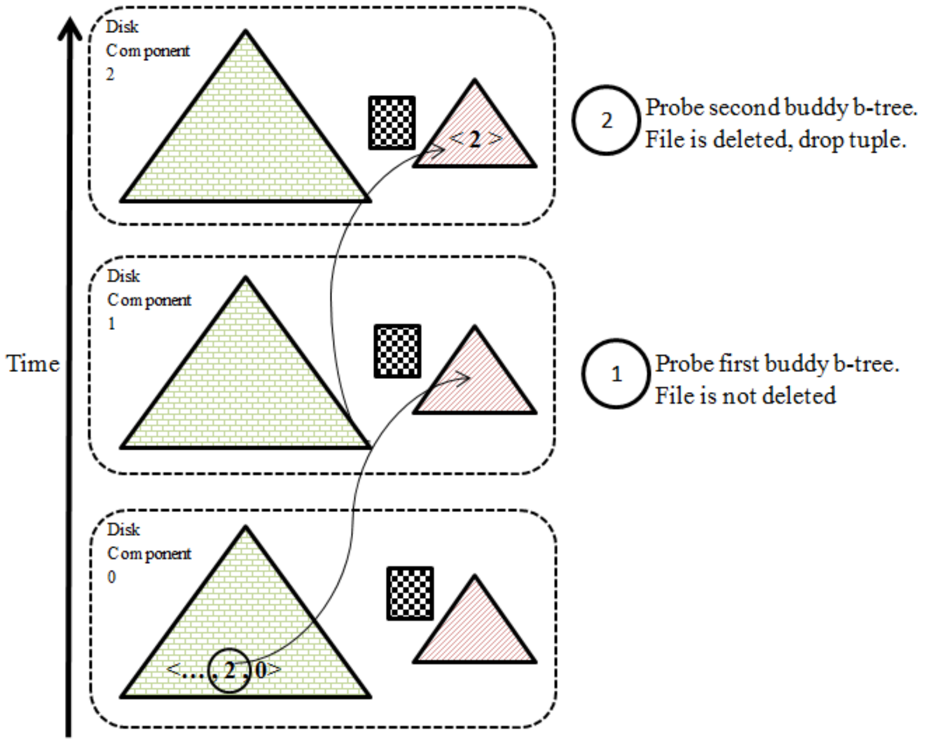
Disk Component 2
< 2 >
2
Probe second buddy b-tree. File is deleted, drop tuple.
Disk Component 1
Time
1
Probe first buddy b-tree. File is not deleted
Disk Component 0
<… , 2 , 0>

Figure 6.10: Search within multiple components of an index

to reduce the number of I/O operations during the search. The buddy B-trees of an index are also used for space reclamation when the index component's merge operation is triggered by its merge policy. Filtering deleted tuples at this early stage of the pipeline has additional benefits since it also reduces the number of tuples to be sorted in the sort operator that precedes the external lookup operator.

If any of the Hyracks jobs involved in a refresh fails, the computed delta is discarded, the cluster controller instructs each node to abort the transaction locally in all the participating indexes, the transaction is marked complete, and an error message with the cause of failure is returned to the end user. If all Hyracks jobs complete successfully, the cluster controller records the state of the transaction as ready-to-commit and then instructs all nodes to commit the transaction locally in its indexes. After that, the cluster controller adds the delta to the ExternalFile dataset and marks the refresh transaction as complete. During AsterixDB's startup bootstrap, the system loops over all ongoing transactions to perform global recovery. In the case of a system crash, any transaction that was in the running state is rolled back, and any transaction that was in the ready-to-commit state is rolled forward. Since refresh operations always bulk load their changes into new components, no record level logging is required.

6.2.6 Consistency and Concurrency

In AsterixDB, a consistency requirement is that an AQL query that accesses an indexed external dataset should access the same version of the dataset throughout its execution. This means that the refresh operation's effect must either be seen by all the operators of a query in all the nodes or not seen by any operator. This behavior must be maintained in the case when a dataset is being refreshed while queries are accessing it using multiple indexes or using a scan operator. We now explain how AsterixDB achieves this and how queries, refreshes, index creates, and index deletes can interleave correctly.

AsterixDB maintains at most two logical versions of each index. The cluster controller keeps track of the number of queries accessing each version of each external dataset. An index version locally maps to a list of disk components that form the index. When a query needs to access an external index, the compiler assigns it the most recent version of the index, and the number of queries known to be accessing that index version is incremented. When a query needs to access the same dataset a second time, the compiler assigns the previously assigned version to the new operator in the same query. When a query finishes execution, the index access count for each of its accessed datasets is decremented. When the number of queries accessing an old version of an indexed dataset reaches 0, the dataset version becomes inactive since no more queries are going to be assigned that version.

Multiple secondary index creation operations for a dataset can be run concurrently. On the other hand, each external dataset refresh statement is serialized with respect to other refresh statements and is mutually exclusive with index create statements. This is critical since index create statements build indexes with respect to a specific snapshot. Index create statements hence need to wait for an ongoing refresh to either succeed or fail before it can proceed with building the index according to the state of the stored snapshot of the dataset. The refresh operation also can't proceed if the older version of the index is still active, so it waits until the last query accessing that version finishes execution. Since we don't keep track of the number of queries accessing each dataset's index, an index can only be deleted when no other operations are running on its dataset.

Locally on each node, an external index maintains a list of pointers to components that map to each version of the index. Since a local index partition has no way of knowing when a version of the index is not needed anymore, the information about the older version of the index is only deleted when the system shuts down or when a new refresh starts. When a query tries to search an index, it carries with it the version number of the index that it was assigned by the compiler, and the search cursor will then only search components that belong to that version. The index component merge policy only considers the most recent version of the index when it is triggered, without dismissing the possibility that - between the start of a merge operation and its completion - a new version of the index could have been added. Upon completion of merge operations, the index takes care of updating its two versions by deleting the merged components and replacing them with the new component. With these control structures, consistency is guaranteed while allowing queries and refreshes to be run concurrently.

Chapter 7

Performance Evaluation

In this chapter, we presents some initial experimental results to show the performance of AsterixDB's external datasets. We use the HDFS adapter on Hive-created data and compare the performance of AsterixDB against Hive itself and also against AsterixDB running with the same data loaded natively into the system. We also show the space performance tradeoffs between moving data into the AsterixDB system and indexing it as external data.

7.1 Experimental Setup

The following experiments are conducted on a cluster of 10 nodes. The nodes are Dell PowerEdge 1435SCs with 2 Opteron 2212HE processors, 8GB DDR2 RAM each, and 2x 1TB 7200 rpm hard drives. The version of Hadoop used in the experiments is 2.2.0 with 10 data nodes each using the two available hard drives. The I/O buffer size for Hadoop is set to 64 KB, and the default block size is set to 64 MB with the default replication factor of 3. The number of HDFS name node handlers is set to 100, allowing upto

10 concurrent RPC calls from each node in the cluster at any time, and the minimum I/O buffer size (which determines the minimum size read by an HDFS reader) is set to 64KB. 2GB of memory is allocated for map tasks and 4GB for reduce tasks, with 2GB of memory being allocated for sorting operations with 200 merge streams. The Hive version used in these experiments is version 0.13.0. An AsterixDB instance is installed on the same cluster, with each node having two partitions and each partition using one of the two I/O devices. The cluster controller is allocated 1GB of memory. In each of the node controllers, the memory allocated for sort and join operations is 512 MB. The memory allocated for disk buffer cache is 1.5GB and the memory allocated for indexes memory components is 512 MB.

Dataset	Format	Size	Number of files
Lineitem	Text	187.2 GB	10
Lineitem	Sequence	207.6 GB	749
Lineitem	RCFile	173.9 GB	749
Orders	Text	41.9 GB	10
Orders	Sequence	46.1 GB	167
Orders	RCFile	39.35 GB	167
Customer	Text	5.75 GB	10
Customer	Sequence	6.22 GB	23
Customer	RCFile	5.6 GB	23

Table 7.1: HDFS raw data files for TPCH

The dataset used with in these experiments is a TPC-H[3] dataset at a 250GB scale value. The experiments use only the Lineitem, Orders and Customer tables. the data is uploaded in its generated text format (delimited text with pipe delimiters) to HDFS and is block-balanced among the 20 disks. External hive tables are created on the uploaded text files and then used to perform a SELECT INSERT into their equivalent Hive tables, thus creating the same datasets stored in Sequence file format and RCFile format. The row group size in the produced RCFile datasets is 4 MB. The sizes of each of the datasets in their different raw formats is shown in Table 7.1.

7.2 Results and Analysis

In our experiments, we focus on evaluating the performance of AsterixDB's external HDFS datasets. We first evaluate the space and time cost of loading the data into an internal dataset in AsterixDB. We then evaluate the performance of AsterixDB's external data access without indexes on all datasets using all of the file formats against the performance of AsterixDB for internal datasets and against Hive's performance on Hive tables on the same data. We then evaluate the cost of building external indexes in AsterixDB, and finally, we evaluate the performance of using AsterixDB with external indexes as compared to AsterixDB's internal storage indexing.

7.2.1 Cost of loading data

Dataset	Loading time	Primary Index Size
Lineitem	43 min	334 GB
Orders	9 min	60 GB
Customer	1.25 min	7.7 GB

Table 7.2: Cost of loading data

Loading data into internal datasets introduces additional costs that are eliminated completely with the use of external datasets. The time and space costs of loading the three datasets into internal AsterixDB datasets are shown in Table 7.2. Loading data into the system involves in addition to fetching and parsing data from external resources, hash partitioning and sorting the records on their primary keys, and then bulk loading them into partitioned B-Tree indexes. Even though a record's binary representation is smaller than its raw format, the overall space cost is larger than the external file size because of the additional space needed by the index structure. Creating external datasets, on the other hand, has negligible space and time costs.

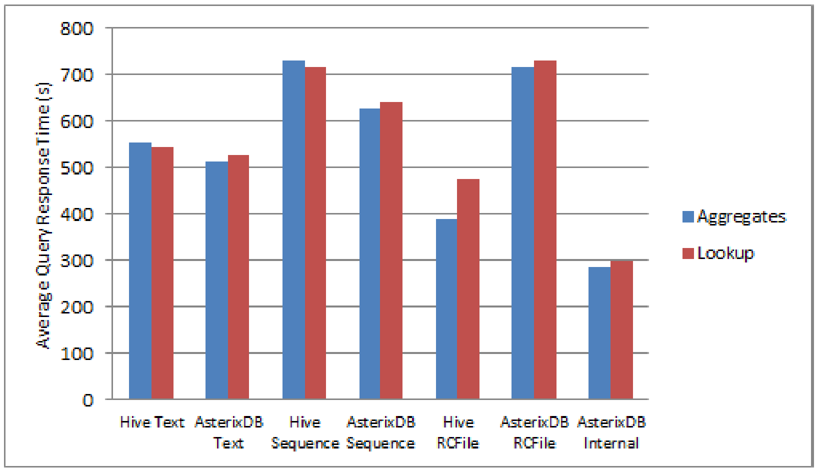

Figure 7.1: Lineitem dataset full scan access

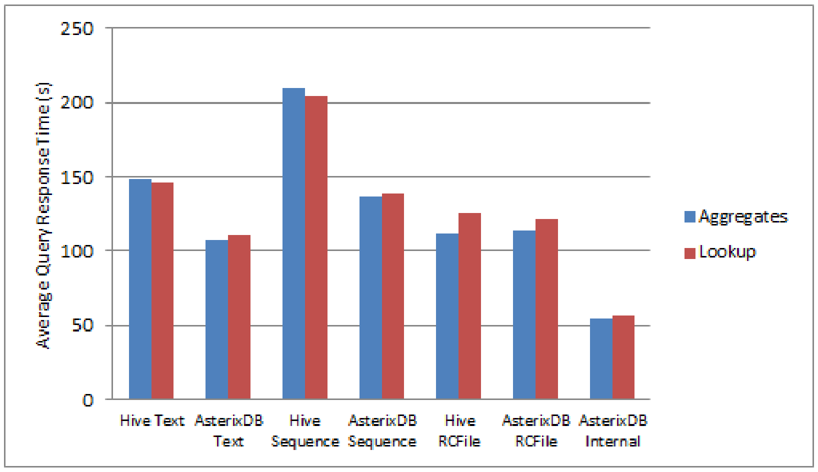

Figure 7.2: Orders dataset full scan access

7.2.2 Full scan performance

In this section, we evaluate the full scan performance on the three datasets. The queries that we ran for the full scan access performance evaluation either calculate simple aggregate values over a single attribute of a dataset or else perform a single record lookup. Examples of these queries in Hive Query language (HQL) and in AQL can be found in Listings 7.1 and 7.2. Looking at the performance in Figures 7.1, 7.2, and 7.3, we can see that AsterixDB's internal datasets are always faster than its external datasets and also faster than Hive for all table formats. This is due to the fact that internal

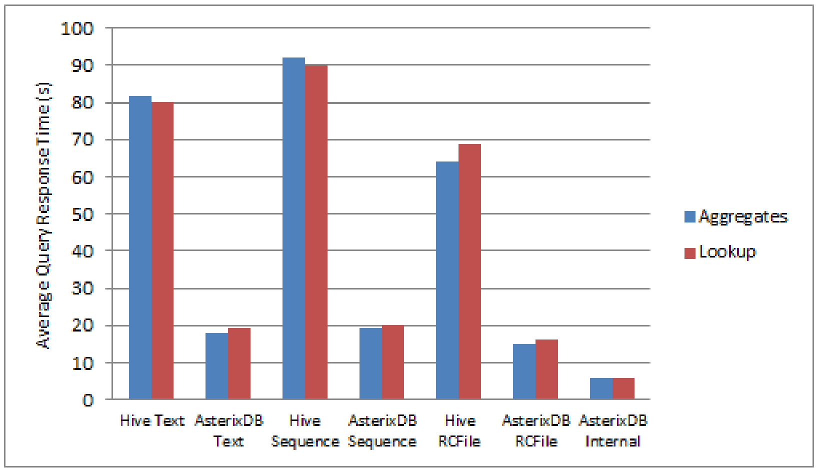

Figure 7.3: Customer dataset full scan access

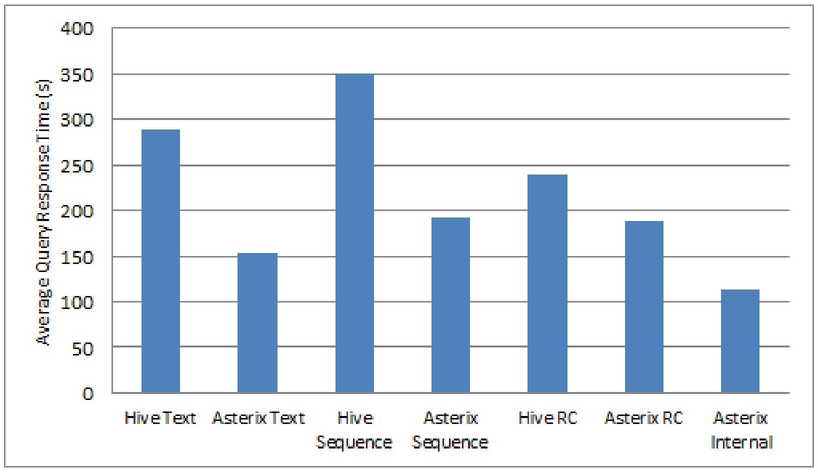

Figure 7.4: Join Orders and Customers on customer key

AsterixDB data is stored in the system's binary data format and doesn't require any parsing. In addition, the amount of data read is smaller since only the leaf nodes of the internal dataset's primary indexes are scanned. Queries on external data stored in Text and Sequence formats run faster with AsterixDB and its external datasets than using Hive itself to query the same data. This can be attributed in part to the lower cost of running Hyracks jobs compared to the cost of running MapReduce operations. However, Hive runs queries on the Lineitem dataset stored in RCFile format faster than AsterixDB's external dataset performance on the same files. This is caused by two-step

```
SELECT MAX(li.L_PARTKEY) FROM lineitem li;
SELECT * FROM lineitem li
  WHERE li.L_SUPPKEY = 559288
  AND li.L_LINENUMBER = 7
  AND li.L_QUANTITY = 5
  AND li.L_EXTENDEDPRICE = 6321.45;
```

Listing 7.1: Full scan query examples in HQL

```
use dataverse tpch;
// find the maximum
let $litems := for $li in dataset Lineitem
  return $li.partkey
return max($litems);
// lookup a specific record
for $li in dataset Lineitem
  where $li.lsuppkey = 559288
  and $li.llinenumber = 7
  and $li.lquantity = 5
  and $li.lextendedprice = 6321.45
return $li;
```

Listing 7.2: Full scan query examples in AQL

parsing, as AsterixDB uses the built-in Hive parser which first deserializes records into Hive objects using Hive serde and then parses them into AsterixDB's internal format. On top of that, AsterixDB's adapters always parses all the fields of each record regardless of whether these fields are needed to answer the query. The negative effect of this two-step parsing is not seen in queries that run on the Orders and Customer datasets due to their smaller sizes, and hence, AsterixDB runs these queries faster than Hive at all times.

We followed these single dataset experiments with queries that perform a filter using customer key on the Orders dataset followed by a join with the Customer dataset on the customer key. The queries used for the join experiments can be found in listings 7.3 and 7.4. The size of the join output is 1010 records and the performance results are shown in Figure 7.4. In addition to AsterixDB's ability to read these datasets faster than Hive, it uses a Grace hash join algorithm which is more efficient than the shuffle

```
SELECT cust.c_custkey,cust.c_name, ord.o_orderkey FROM orders ord JOIN
    customer cust ON (ord.o_custkey = cust.c_custkey)
WHERE ord.o_custkey > 1463524
and ord.o_custkey < 1463624
```

Listing 7.3: Join query in HQL

```
use dataverse tpch;
for $ord in dataset Orders
 for $cus in dataset Customer
  where $ord.ocustkey = $cus.ccustkey
  and $ord.ocustkey > 1463524
  and $ord.ocustkey < 1463624
return {
"custkey":$cus.ccustkey,
"name":$cus.ccname,
"orderkey":$ord.oorderkey};
```

Listing 7.4: Join query in AQL

join used by Hive.

7.2.3 Cost of indexing datasets

The next set of experiments evaluates the cost of creating indexes over external datasets against their internal equivalents. Table 7.3 shows these results. The results show that the time needed for indexing external datasets is always more than the time needed for indexing internal ones. This is due to the additional I/O cost, the parsing cost and having composite Record IDs (which slows down the sort operation). In addition to that, the HDFS readers at the indexing source operator at the beginning of the pipeline in Figure 6.6 might scan files from partitions residing in the same I/O device in some cases, which creates added delay. This is caused by a limitation in the current Hadoop API, which doesn't provide a mechanism to determine the I/O device hosting each block, at the time of writing this thesis. The size of the indexes for RCFile data is larger than the same indexes on other formats due to the additional row number field in each RID.

TPCH-Dataset	Index Key	Indexing Time(s)	Resulting Index Size
Text Lineitem	order key	1458	48.8 GB
Sequence Lineitem	order key	1678	48.8 GB
RCFile Lineitem	order key	1796	58.6 GB
Internal Lineitem	part key	1313	40 GB
Text Lineitem	part key	1632	48.8 GB
Sequence Lineitem	part key	1802	48.8 GB
RCFile Lineitem	part key	2156	58.6 GB
Internal Orders	customer key	297	7.7 GB
Text Orders	customer key	356	12 GB
Sequence Orders	customer key	422	12 GB
RCFile Orders	customer key	473	14.5 GB
Text Orders	order key	300	48.8 GB
Sequence Orders	order key	362	48.8 GB
RCFile Orders	order key	341	58.6 GB
Text Customer	customer key	38	1 GB
Sequence Customer	customer key	39	1 GB
RCFile Customer	customer key	49	1.47 GB

Table 7.3: Indexing comparison between external and internal datasets

7.2.4 Index access performance

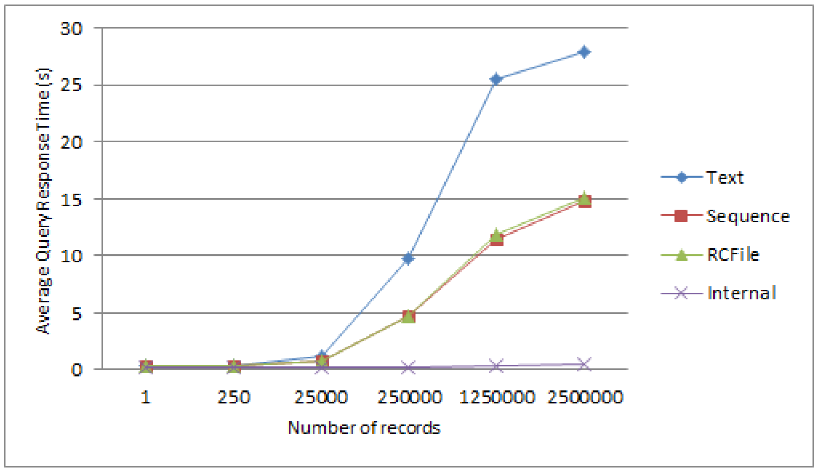

Figure 7.5: Lookup queries on Orders dataset using order index

We now evaluate index access on external and internal datasets. First, we compare the performance of the secondary index access on external datasets with the primary index access on internal dataset. We then evaluate the performance of the secondary index

```
use dataverse tpch;
count(
  for $ord in dataset TextOrders
    where $ord.oorderkey > 1000000
    and $ord.oorderkey < 1500000
  return $ord
    );
```

Listing 7.5: Lookup query on Orders dataset using the order key index

access on different external and internal datasets. We finally compare the performance of index nested-loop join queries over external and internal datasets.

The first set of index access queries compare the performance of external secondary indexes on the Orders datasets of different formats on the order key against the internal Orders dataset's primary index. An example of these queries can be seen in Listing 7.5. The external raw Text files in HDFS have the records sorted on the index key, while the data in the original 10 files of this dataset were split into 167 files when inserting them into Hive tables. The data internally is hash-partitioned on the order key attribute values. Figure 7.5 shows the results of running these queries. We can see that for small ranges all dataset formats perform about the same. This is due to the cost being dominated by the Hyracks job starting cost. As the range grows larger, we can see that for the internal dataset, the low cost is maintained. This behavior is expected since it essentially performs a single index search using the dataset's primary index. The load balancing is much better for the internal dataset as wel, as records are hash-partitioned on the search key. In addition to the load balancing advantage of the internal dataset, the external datasets are first searching the secondary indexes, then sorting the produced tuples, and then performing external data seek, read, and parse operations. An interesting observation that shows the effect of the record organization in the external source is the gap between the Sequence and RCFile datasets on one hand and the Text dataset on the other. Looking at the logged load distribution, we found

```
use dataverse tpch;
count(
  for $ord in dataset TextOrders
    where $ord.ocustkey > 3000000
    and $ord.ocustkey < 3100000
  return $ord
    );
```

Listing 7.6: Lookup query on Orders dataset using customer key index

that having the records totally ordered in the original Text files created a clear imbalance which affected the index access performance negatively. This property was not preserved when the data was loaded using Hive into the other two datasets, and hence, their index access performed better than the the index access for the Text formatted data in this case.

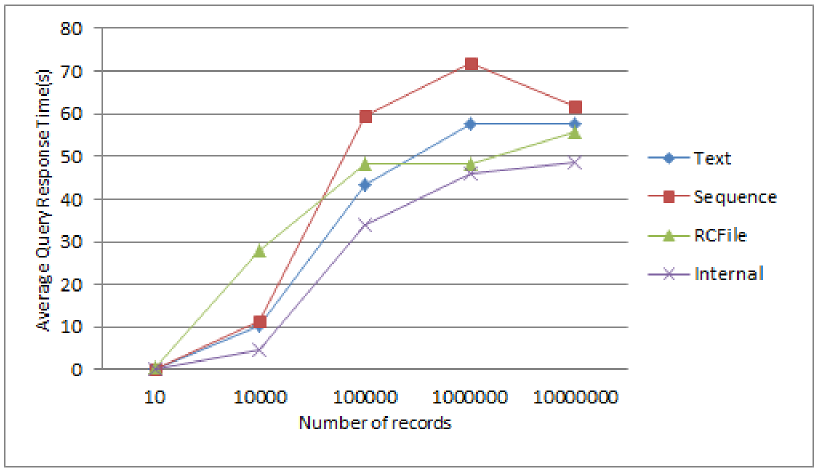

Figure 7.6: Lookup queries on Orders dataset using customer index

The next set of external index queries compare the performance of secondary indexes on both internal and external datasets. An example of these queries that perform records lookup using the customer key on the Orders datasets can be found in Listing 7.6. The results, seen in Figure 7.6, show roughly comparable performance for all types of datasets. The workload distribution was similar for all the datasets in this query since the records are not ordered on the customer key for any of them. At around the 10,000 range, the RCFile dataset's secondary index access suffers from the cost of additional

I/O because it reads a complete row group (about 40,000 records) if any records of interest are stored in that row group, and so it becomes less performant than the other datasets. At around the 100,000 range, the RCFile's dataset I/O cost started becoming lower again since more records were found in the same row groups and the benefits of performing sequential reads in this case outweighed the cost of reading extra records from disk. Overall, this experiment shows that secondary external indexes perform quite well as compared to internal ones.

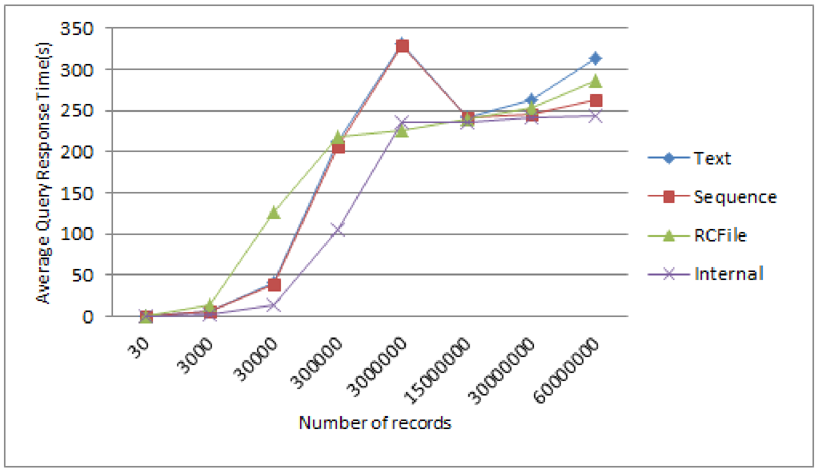

Figure 7.7: Lookup queries on Lineitem dataset using part index

To confirm these findings, similar lookup queries were also run on the Lineitem datasets using the partkey secondary index. The results are shown in Figure 7.7. These queries exhibit similar behavior until the 3,000,000 range, where queries over RCFiles datasets using secondary indexes outperformed internal dataset's queries. This behavior can be attributed to having more records of interest grouped and read together using sequential I/O reads which provided some speedup. In this range, queries that access Text and Sequence data performed much worse than the same queries on the internal or the RCFile dataset. Continuing until the range of 60,000,000 records, we can see that the performance of secondary index access for all formats of the dataset provided similar performance results.

```
use dataverse tpch;
count(
  for $ord in dataset Orders
   for $cus in dataset Customer
    where $ord.ocustkey =  /*+ indexnl */ $cus.ccustkey
    and $ord.ocustkey > 1463524
    and $ord.ocustkey < 1463624
  return $cus
  );
```

Listing 7.7: Index nested-loop join query in AQL

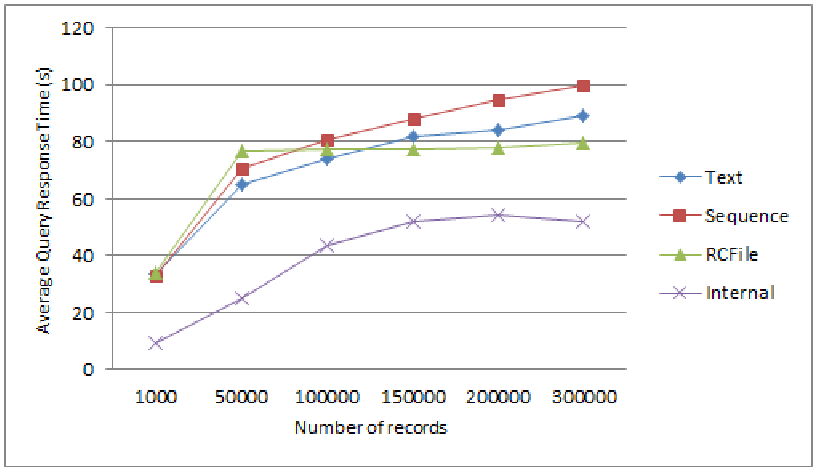

Figure 7.8: Join Orders and Customers datasets on customer key

The last set of queries perform a select on the Orders dataset using a condition on the customer key followed by an index nested-loop join with the Customer dataset using the customer key. An example of the join query can be found in Listing 7.7. The first observation seen in Figure 7.8 is that the internal dataset joins are faster than the external ones. This is caused by a number of contributing factors. The records produced from the first index search on the Orders dataset are each sent to a specific node which holds the customer record with the matching customer number in the internal datasets case. For the external datasets, each record must be broadcast to all the nodes since the matching record's location is unknown. On top of that, at the join side, the internal dataset only searches on a primary key, while for the external datasets, the search is performed on a secondary index and must be followed by seek, read, and parse operations to fetch the record itself. Nonetheless, the performance of the index nested-loop joins also show the potential benefits of having indexing over external data.

Chapter 8

Conclusions and Future Work

In this thesis, we have described how AsterixDB addresses the need for external data access for users with existing external data. We explained how AsterixDB uses different adapters to read data from different sources and formats and how it performs on the fly parsing of records in an efficient way. We also explained the existing need to improve external data access performance and how we addressed this need by building incrementally refreshable external indexes. We then explained the data access semantics when indexes are built on external data and how AsterixDB preserves consistency among different access methods in the face of a changing external dataset. We also described some of the key implementation details for the index building, index access, and index refresh operations. We showed how the system deals with failures and how it maintains a consistent state of indexed datasets at all times. We followed that by describing the control structures used to ensure consistency and concurrency during different operations.

This thesis also includes a performance evaluation for different access methods for external HDFS data. Through these performance experiments, we showed that AsterixDB's HDFS dataset access provides good full scan performance for different popular file formats. We also showed the cost of building indexes on external datasets compared to building indexes on internal datasets. Finally we ran different index access queries on different datasets and showed that indexing of external data provides substantial performance improvements and even competes respectably with the performance of internal secondary index access.

The current design of the system allows reusability of most of the existing components which enables supporting the indexing of more types of external data in the future. Suggested future work to improve the current external data access include pushing field projection to the different external data access operators and indexing more types of external data in HDFS and in local file systems. With the current design, the requirement for indexing an additional HDFS input format is the identification of its RIDs, extracting them while reading files' splits and using them to look records up. As for data in local file systems, additional changes would be required.

Bibliography

[1] Apache Hadoop. http://www.hadoop.org/.

[2] AsterixDB. http://asterix.ics.uci.edu/.

[3] TPC-H Website. http://www.tpc.org/tpch/.

[4] Oracle external tables concepts, Apr 2014.

[5] A. Abouzied, D. J. Abadi, and A. Silberschatz. Invisible loading: Access-driven data transfer from raw files into database systems. In *Proceedings of the 16th International Conference on Extending Database Technology*, EDBT '13, pages 1–10, New York, NY, USA, 2013. ACM.

[6] I. Alagiannis, R. Borovica, M. Branco, S. Idreos, and A. Ailamaki. Nodb: Efficient query execution on raw data files. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 241–252, New York, NY, USA, 2012. ACM.

[7] V. Borkar, M. Carey, R. Grover, N. Onose, and R. Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE '11, pages 1151–1162, Washington, DC, USA, 2011. IEEE Computer Society.

[8] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[9] J. Dittrich, J.-A. Quiané-Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad. Hadoop++: Making a yellow elephant run like a cheetah (without it even noticing). *Proc. VLDB Endow.*, 3(1-2):515–529, Sept. 2010.

[10] M. Y. Eltabakh, F. Özcan, Y. Sismanis, P. J. Haas, H. Pirahesh, and J. Vondrak. Eagle-eyed elephant: Split-oriented indexing in hadoop. In *Proceedings of the 16th International Conference on Extending Database Technology*, EDBT '13, pages 89–100, New York, NY, USA, 2013. ACM.

[11] Y. He, R. Lee, Y. Huai, Z. Shao, N. Jain, X. Zhang, and Z. Xu. Rcfile: A fast and space-efficient data placement structure in mapreduce-based warehouse systems. In

Proceedings of the 2011 IEEE 27th International Conference on Data Engineering, ICDE '11, pages 1199–1208, Washington, DC, USA, 2011. IEEE Computer Society.

[12] S. Idreos, I. Alagiannis, R. Johnson, and A. Ailamaki. Here are my data files. here are my queries. where are my results? In *CIDR*, pages 57–68. www.cidrdb.org, Apr 2011.

[13] M. Islam, A. K. Huang, M. Battisha, M. Chiang, S. Srinivasan, C. Peters, A. Neumann, and A. Abdelnur. Oozie: Towards a scalable workflow management system for hadoop. In *Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies*, SWEET '12, pages 4:1–4:10, New York, NY, USA, 2012. ACM.

[14] A. Jain, A. Doan, and L. Gravano. Optimizing SQL Queries over Text Databases. In *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, ICDE '08, pages 636–645, Washington, DC, USA, 2008. IEEE Computer Society.

[15] K. Lorincz, K. Redwine, and J. Tov. Grep versus FlatSQL versus MySQL: Queries using UNIX tools vs. a DBMS, 2003.

[16] J. Melton, J.-E. Michels, V. Josifovski, K. Kulkarni, P. Schwarz, and K. Zeidenstein. SQL and Management of External Data. *SIGMOD Rec.*, 30(1):70–77, Mar. 2001.

[17] C. Mohan, B. Lindsay, and R. Obermarck. Transaction Management in the R* Distributed Database Management System. *ACM Trans. Database Syst.*, 11(4):378–396, Dec. 1986.

[18] Raman Grover, Michael J. Carey. Scalable FaultTolerant Data Feeds in AsterixDB, 2014.

[19] Sattam Alsubaiee, Alexander Behm and Vinayak Borkar, Zachary Heilbron and Young-Seok Kim, Michael J. Carey, Markus Dreseler and Chen Li. Storage Management in AsterixDB . volume 7, pages 841–852. VLDB Endowment, June 2014.

[20] Sattam Alsubaiee, Yasser Altowim, Hotham Altwaijry, Alexander Behm,Vinayak Borkar, Yingyi Bu, Michael Carey, Khurram Faraaz, Eugenia Gabrielova, Raman Grover, Zachary Heilbron, Young-Seok Kim, Chen Li, Guangqiang Li, Ji Mahn Ok, Nicola Onose,Pouria Pirzadeh, Vassilis Tsotras, Rares Vernica, Jian Wen, Till Westmann, Inci Cetindil, and Madhusudan Cheelangi. AsterixDB: A Scalable, Open Source BDMS. Submitted for publication, 2014.

[21] Y. Xu, P. Kostamaa, and L. Gao. Integrating hadoop and parallel dbms. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 969–974, New York, NY, USA, 2010. ACM.