External Data Access and Indexing in AsterixDB

Abdullah Alamoudi, Raman Grover, Michael J. Carey, Vinayak Borkar

Dept. of Computer Science,
University of California Irvine, CA, USA - 92697
{alamouda, ramang, mjcarey, vborkar}@uci.edu

ABSTRACT

Traditional database systems offer rich query interfaces (SQL) and efficient query execution for data that they store. Recent years have seen the rise of Big Data analytics platforms offering query-based access to "raw" external data, e.g., file-resident data (often in HDFS). In this paper, we describe techniques to achieve the qualities offered by DBMSs when accessing external data. This work has been built into Apache AsterixDB, an open source Big Data Management System. We describe how we build distributed indexes over external data, partition external indexes, provide query consistency across access paths, and manage external indexes amidst concurrent activities. We compare the performance of this new AsterixDB capability to an external-only solution (Hive) and to its internally managed data and indexes.

Categories and Subject Descriptors: H.2 [DATABASE MANAGEMENT]: Systems - Query Processing

Keywords: AsterixDB, External data, HDFS, Access, Indexing.

1. INTRODUCTION

Database management systems employ many techniques to achieve good performance. These include storage structures, access methods, caching, and efficient query execution. As a prerequisite, a DBMS requires data to be stored into its storage layer and modified through its interface. Having to pre-load data can be a major obstacle, particularly when data is being produced in huge quantities by multiple sources and being persisted in different storage systems and formats. Providing full-scan access to external data from a DBMS is a first step that enables the use of queries rather than error-prone ad hoc analysis scripts. However, the lack of indexes is bound to give unacceptable query response times. A natural next step is to support indexing for external data.

CIKM'15, October 19–23, 2015, Melbourne, Australia.

DOI: http://dx.doi.org/10.1145/2806416.2806428.

1.1 External Access and Indexing Challenges

Efficient and flexible external data access in a parallel data manager involves challenges stemming from lack of control over the representation of records, their storage locations, and the data modification path(s). These include:
C1) *Disparate Data Sources and Formats*: An external access facility must be generic and extensible to allow access to data in different data sources and formats.
C2) *Seamless Integration*: The distinction between external and internal data should not be visible to the end-user when framing queries in the language offered by the system.
C3) *Parallel/Efficient Access*: A parallel data manager must exploit parallelism in accessing external data and utilize distributed index structures to achieve load balancing.
C4) *Maintenance of Indexes*: External data may change, causing indexes to become stale. The system must enable users to refresh the indexes with *transactional* behavior.
C5) *Consistency of Query Results*: The system should offer consistent query results for external data regardless of the access path used and concurrent changes to external data.

1.2 Contributions

We describe how we have recently enhanced AsterixDB to enable users to efficiently query externally stored data. Our solution supports a set of popular external data source types and formats but is extensible to cater to new data sources and formats. It allows users to build multiple distributed indexes (e.g., B+ Trees and/or R-Trees) over external data. The query compiler then utilizes these indexes to accelerate queries. We offer the following contributions:
(1) *External Data Access and Indexing*: We provide a complete conceptual and technical design for providing access to and building distributed indexes over external data.
(2) *Access Semantics*: We introduce *external data snapshot* semantics for queries accessing indexed external data. We describe mechanisms to keep indexes in known synchrony with the data and provide consistent query results.
(3) *Incremental Updates*: We detail the design for refreshing a dataset's indexes incrementally in AsterixDB in an efficient and robust manner.
(4) *Contribution to Open-Source*: AsterixDB is open-source. Its support for adapters, parsers, and indexing is extensible to allow other project contributors to add new adapters and parsers for other external data sources and formats.
(5) *Experimental Evaluation*: We measure the external data access performance for different HDFS formats, comparing AsterixDB's performance when data resides externally in Hive tables [2] to that of Hive itself. We also compare per-

formance for internal versus external data to see what can be attained without moving data into AsterixDB.

The rest of the paper is organized as follows. Section 2 covers related work. We review AsterixDB and the Hadoop Distributed File System (HDFS) in Section 3. Section 4 introduces the language-level support and concepts related to external data access. We detail the corresponding design and implementation in Section 5. Section 6 presents experimental results, and Section 7 concludes the paper.

2. RELATED WORK

The need to access external data has been long recognized in the database community [16] and was added to the SQL standard. Leading DBMSs provide raw data scanning capabilities via features such as External Tables in Oracle and MySQL and External Link and Open Row Set in MS SQL. For extensibility, Oracle and DB2 offer Table Functions and MS SQL provides Table-Valued User Defined Functions. Use of these features involve full scans over external data, which perform worse than queries over internally stored data. In contrast to using a DBMS to access and query external data, [15] described the use of Unix tools (awk, grep) as an alternative, introducing FlatSQL as a language to operate over text files. The trade-offs involved in using Unix tools when accessing external data were further explored in [14] and influenced the "NoDB" system [4].

It is common today to have large data files that span multiple machines in a distributed file system such as HDFS. Loading such data into a DBMS imposes significant costs. Several efforts (e.g. [18]) have focused on integrating Hadoop (Map-Reduce) with a DBMS to avoid the costs involved in movement of data. HadoopDB [3] allowed running queries on raw HDFS files using MapReduce and provided support for incrementally loading data into the DBMS's storage layer. Other efforts to improve query execution over data in HDFS include the use of Split-oriented indexes [12] to prune unwanted input splits to reduce I/O costs and building Trojan indexes [11] into the physical file splits when loading data into HDFS storage using user-defined functions. In the High Performance Computing field, SciHadoop field[9] provide different mechanism to prune HDFS data partitions and avoid the full scan cost. More recent related work can be found in [7] and [10].

Our approach to external data access is unique in several ways. It allows users to build distributed record-level indexes over external data. It doesn't cache or re-write data, the query results are access path independent, and data is not loaded into the AsterixDB storage layer at any time. The presence of indexes doesn't affect the external data, and we support indexing of multiple HDFS input formats. This is accomplished largely by making externally stored data "look" like internally stored data from the perspective of most of the system's query processing components.

Our work has key differences from the concurrent effort to index HDFS data in Polybase [13]. The differences, detailed later in the paper, include a location-aware index partitioning strategy, a Record Id (RID) optimization, the exploitation of AsterixDB's Log-Structured Merge (LSM) trees when updating indexes atomically, support for indexing RCFiles, snapshot-based consistency of query results, and the roles of MapReduce (in Polybase) and HDFS in query processing.

3. BACKGROUND

We begin by briefly reviewing AsterixDB and HDFS, the systems for which this work was designed, implemented, and evaluated.

AsterixDB: AsterixDB is an Apache incubator open source Big Data management system. It has its own flexible data model – ADM (inspired by JSON) – and comes with a full query language – AQL – for querying and analyzing semi-structured data. Its shared-nothing architecture involves a central Cluster Controller (CC) and a set of *worker* nodes referred to as Node Controllers (NCs). Data resides as records in datasets that are hash-partitioned on their primary keys and stored in primary B+ tree indexes. Secondary indexes (when created) are co-located with the corresponding partitions of the primary index [5]. To execute queries, AsterixDB uses Hyracks [8] as its execution engine. A Hyracks job is a DAG built using data *operators* and *connectors*.

HDFS: HDFS is the distributed file system of Hadoop [1]. HDFS files are partitioned into replicated binary blocks to enable parallel reads and hardware failure tolerance. HDFS consists of a *NameNode* that stores files' metadata information and a set of *DataNode(s)* that store the binary blocks of data. HDFS files are append-only, meaning that records contained in a file cannot be modified.

4. EXTERNAL DATASETS

External datasets in AsterixDB enable users to query data stored in external sources. Unlike internal datasets, external datasets do not support AQL *load*, *insert*, or *delete* statements. External sources may store data in a variety of formats and have specific protocols for fetching data. To cater to the diversity of data sources and formats, a data manager must also be extensible.

4.1 Dataset Adapters

Connecting to an external data source and receiving, parsing, and translating its data into binary ADM records (for AsterixDB processing) is done by a *dataset adapter*. The use of an adapter includes providing a set of parameters that it uses when interfacing with the source. AsterixDB offers a set of initial adapters for common data sources; its design also offers a plug-and-play model to allow new adapters to be added. Common external data sources and their handling by their respective AsterixDB adapters include:

(1) **Data Files in Local File Systems**: Files residing in the local file systems of the AsterixDB nodes are accessible using the Local File System adapter. This adapter is configured with a set of file identifiers, where an identifier combines the host name (AsterixDB node) and absolute path of a local file on the host. The format associated with the file is additionally specified. AsterixDB's built-in text parsers support parsing of data that is in delimited, JSON, or ADM format. AsterixDB also allows using a custom implementation of a parser for additional formats. Multiple files on a given node or on different nodes are read and parsed in parallel to exploit node-level and cluster-level parallelism.

(2) **Data Files in HDFS**: The HDFS adapter performs the same function for data files residing in HDFS. This adaptor takes an additional argument that describes the InputFileFormat of the dataset's files.

(3) **Data in Web Resources**: AsterixDB's web adapters read data from resources identified by URIs. The level of parallelism when using this adapter is dictated by the num-

```
create external dataset ExLineitem(lineitemType)
using hdfs (
("hdfs"="hdfs://namenode: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 1: Creating and querying an External Dataset

```
create index OrdIdx on Lineitem(lorderkey);
create index OrdIdx on ExLineitem(lorderkey);
```

Listing 2: Creating a B+ Tree Index over Internal and External datasets

```
refresh external dataset ExLineitem;
```

Listing 3: Refreshing an External Dataset

ber of URIs, and the task of reading and parsing records is assigned to different nodes during query compilation.

An adapter is referred to in AQL by its *alias* and specified as part of the *create external dataset* statement. The example statement in Listing 1 uses the HDFS adapter and provides parameters to describe the HDFS instance and contained data. There is no subsequent distinction between internal and external datasets when using them in AQL queries.

4.2 Indexing of External Datasets

The current version of AsterixDB supports full-scan access to external data. The extensions detailed here add support for indexing a variety of HDFS-resident data formats and allow users to build distributed B+ tree and/or R-tree indexes to quickly access records without having to load them into AsterixDB's storage. They were designed to fit in with the typical data lifecycle for a large enterprise Hadoop system, which involves rolling-in and rolling-out data batches in HDFS files. The AQL syntax for defining an index over a set of attributes for any dataset is illustrated in Listing 2. The create index statement syntax is common to both kinds of datasets, internal or external. However, unlike an internal dataset (Lineitem), the data referenced by an external dataset (ExLineitem) can change outside the purview of AsterixDB, causing indexes to become stale. We now explain the consistency model we provide for indexed external data.

4.3 User Experience and Semantics

Manipulation of data in an external source impacts the referencing queries within AsterixDB in two ways. First, external data could change during query execution. Second, creating secondary indexes at different points in time could yield inconsistencies across indexes. To obtain consistent access semantics for external data, we introduce the concept of an *external dataset metadata snapshot*. An external dataset metadata snapshot denotes the state of an external dataset's files at a given point in time. The information contained in a snapshot is used in preserving consistency across an external dataset's indexes and in enforcing a consistent shared view over data between different access paths (full scans or alternative index-based access paths).

The first use of a *create index* statement (Listing 2) with an external dataset implicitly creates a *metadata snapshot* for the external dataset. A metadata snapshot contains only a list of files' absolute paths, sizes, and modification times, and is thus lightweight. It is used to guide subsequent dataset access or index creation. Records added to the external source after a snapshot will remain hidden from AsterixDB until the snapshot and indexes are re-synchronized with the external data via an explicit *refresh external dataset* statement (Listing 3). The refresh operation is executed in a distributed way over an AsterixDB cluster with transactional semantics; in the event of a failure during a refresh operation, the metadata snapshot and all indexes associated with the external dataset are restored to their previous state. We ensure that a refresh statement does not impact concurrent queries and that an executing query always uses the same metadata snapshot associated with an external dataset.

Our user-guided snapshot consistency model differs from the consistency model of Polybase [13]. Polybase computes a metadata delta for HDFS files before answering queries and it aims to deliver results based on the current state of indexed HDFS files. To do so, it resorts to using a more costly Hybrid access path when an index is found to be stale. We avoid this overhead and provide older consistent results.

5. IMPLEMENTATION DETAILS

We now turn to the physical aspects of creating and maintaining snapshots and indexes and using them for queries.

5.1 Unindexed External Data

Consider the example query from Listing 1. In the absence of indexes, the query involves a full scan of the external dataset ExLineitem. The compiler first refers to the Metadata to find the associated dataset adapter and its parameter values. Access to an external dataset is done by an *ExternalDataScan* operator. Multiple instances of this operator may run in parallel, each fetching and parsing records from the external source. A dataset adapter has an associated *factory* class used by AsterixDB to create instance(s) of the adapter. The factory class uses configuration parameters to determine the degree of parallelism (*count constraint*) and any specific set of AsterixDB nodes (*location constraint*) where instances of the adapter should run. As an example, the factory for the HDFS adapter connects with the HDFS NameNode to determine the count and location of input partitions. In determining the location constraint, it gives preference to co-located AsterixDB nodes to facilitate local reads.

5.2 Index Design and Implementation

A parallel external indexing facility needs a way to identify records, to distribute index components in a cluster, and to maintain any/all information in a way that minimizes inter-node communication when an index is used at runtime.

5.2.1 Identifying Records

Indexing requires each data record to have a unique id, hereafter referred to as its RID. (For internal data, the primary key serves as the RID.) RIDs must enable fast

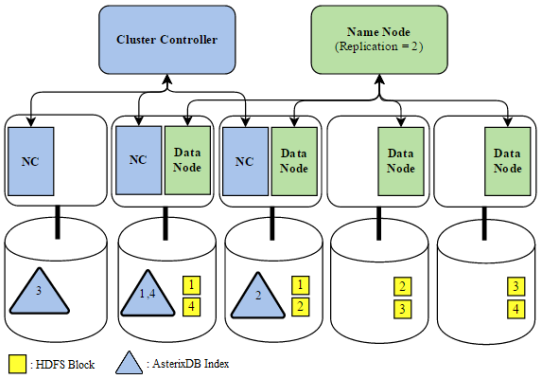


Figure 1: External Data and Index Distribution

record retrieval and must be small in size. As a single external dataset can map to multiple external files, an external record's RID is a combination of a *file id* and a *record locator*. The file id consists of the file's path and modification time. Discovering a file with the same path but a different modification time implies a deletion of the file followed by a creation of a new file. Since storing the file path and modification time in each RID in an index would be very expensive, we replace the long file id with an assigned 4 byte (*integer*) value and store the file id mappings for external datasets' files in a separate B+ Tree index referred to as the *snapshot index*.

The other component of an RID, the record locator, depends on the format of the HDFS file. For the TextInput and SequenceInput formats, the record locator is just the 8 byte (*long*) offset of the record within the file[1]. The RCFile format stores records in columnar order and groups them into row groups of configurable size. Its record locator is then a combination of the 8 byte (*long*) row group offset and 4 byte (*integer*) row number within the containing group.

Support for an additional HDFS file format can be achieved by providing three components. The first is the structure of the RID for the file format; this is used by AsterixDB's query compiler when utilizing operators for index-building or index-accessing jobs. The second component is a runtime indexing adapter that produces RIDs for records being read from HDFS. The third is a runtime lookup adapter that uses the stored RIDs to read the selected records.

5.2.2 Index Distribution

Distribution of data is critical to load-balancing and scalability in a parallel data manager. Records contained in an internal dataset in AsterixDB are hash-partitioned by primary key and stored in primary B+tree indexes across the nodes in a cluster. This achieves near-uniform distribution of records and allows optimizations of joins involving primary keys. Secondary index partitions of internal AsterixDB datasets are co-located with their primary index partitions, which ensures zero extra communication between nodes when accessing data through a secondary index.

External files in HDFS are partitioned into blocks that are replicated and distributed across a set of Data Nodes. Given this, our external indexes use a block-based distribution strategy aimed at achieving locality and load balancing. Co-locating indexes with data reduces communication and allows accessing of HDFS records directly in a short-circuited manner without going through HDFS Data Nodes. The HDFS replication factor and the overlap between AsterixDB NCs and HDFS Data Nodes affect the choices made when selecting an index partition for records in HDFS blocks.

An AsterixDB cluster and an HDFS cluster with data to be externally accessed can have different degrees of node overlap. They may overlap completely, not at all, or partially (the general case, shown in Figure 1). Copies of an HDFS data block may thus be stored on an HDFS node that also hosts an AsterixDB NC (blocks 1, 2, and 4 in Figure 1) or is outside the AsterixDB cluster (block 3 in Figure 1). This yields an index responsibility assignment optimization problem whose objective is to maximize the number of local assignments while minimizing the variance in the number of blocks assigned to each NC. To solve this, we give priority to locality and use a greedy 2-pass algorithm to pick the location of each block's index partition. We first identify the locations of the targeted HDFS blocks and keep track of the number of assignments per NC. In the first pass, we loop over the blocks, assigning overlapping blocks to a co-located NC with the lowest number of assignments (skipping remote blocks). In the second pass, we loop over any unassigned blocks and assign them to the NC with the lowest number of assignments. Ties are broken using node ids.

5.2.3 Snapshot Management

AsterixDB stores metadata about the HDFS files of indexed external datasets; it is used to maintain consistency when interacting with an external dataset. For each external file, the path, modification time, and size in bytes are used to remember its external state at a given point in time and are collectively referred to as the metadata snapshot. (Including the file size is critical because an HDFS file can, in some cases, grow without changing its modification time.) When creating an external index or accessing an external dataset using the *ExternalDataScan* operator, any records lying beyond the stored size of an external file are ignored (treated as not yet visible) to ensure a consistent view of the dataset.

The metadata snapshot of an external dataset is captured from HDFS when the first index on the dataset is created. This snapshot is subsequently updated in a transactional manner when performing external dataset refresh operations. The information contained in the snapshot is stored in AsterixDB's Metadata in a special dataset, the *ExternalFile* dataset. External files are assigned integer ids before capturing their information in a snapshot; these ids are used in the RIDs of external records in place of file paths and modification times. To perform external index accesses, AsterixDB nodes need to lookup metadata snapshots using these assigned ids. This could cause an additional communication overhead and become a bottleneck when accessing AsterixDB Metadata. To avoid this, a *snapshot index* is created for each external dataset in an AsterixDB node. This snapshot index contains the metadata records of external files that belong to the index's associated dataset and it is used to lookup files for indexed access.

5.2.4 Constructing an Index

We now describe the runtime for the *create external index* statement (Listing 2) and the operators involved in the gen-

[1]This is reminiscent of NoDB's positional map[4].

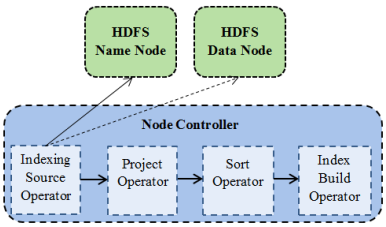


Figure 2: Dataflow for Index Construction

erated Hyracks job. When a user creates a secondary index over an external dataset, the Cluster Controller (CC) first checks whether a metadata snapshot for the dataset exists in the ExternalFile dataset. If one is not found, the CC queries the HDFS NameNode to get the status of the dataset's files and stores it in the ExternalFile dataset. The captured snapshot is then broadcast to all participating nodes in the AsterixDB cluster and stored in B+ tree indexes. The CC then uses the intersection of the stored metadata and the existing files in HDFS to create a list of HDFS block-sized logical file splits to be indexed. Indexing of records in different blocks is assigned to different nodes in AsterixDB as per the distribution strategy described earlier in Section 5.2.2.

A Hyracks load pipeline consisting of four operators is constructed on each participating NC (Figure 2). The *IndexingSource* operator at the head of the pipeline uses an indexing adapter that fetches records and their RIDs. This indexing operator queries the HDFS NameNode to locate the assigned blocks. For blocks local to an NC (e.g., blocks 1, 2 and 4 in Figure 1), the operator can read records directly from the node's disk without communicating with the DataNode as illustrated by the optional dotted connection in Figure 2. For remote blocks (e.g., block 3 in Figure 1), the operator reads records from one of the DataNodes holding the block.

Output from the *IndexingSource* operator is fed into a *Project* operator that extracts the secondary keys from the parsed records and passes them with their RIDs to the *Sort* operator. Tuples sorted on the secondary keys are consumed by the *IndexBuild* operator, which builds the index tree in a bottom up fashion. When the job completes execution on all nodes the index can be used to access the dataset.

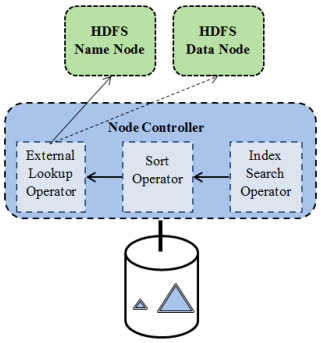


Figure 3: Dataflow for Index Use in Queries

5.2.5 *Using an Index*

When querying an indexed external dataset, AsterixDB's compiler can choose to access the dataset using a secondary index. To access an external dataset through an index, the compiler produces a Hyracks job that contains the pipeline shown in Figure 3. The *IndexSearch* operator uses search predicates to search the secondary index, producing RIDs that point to external records. These RIDs are then sorted by the *Sort* operator and fed to an *ExternalLookup* operator. Sorting RIDs prior to accessing records reduces the associated I/O cost due to sequential access when the number of records are large and incurs a negligible cost when accessing few records since the sort operation would take place in memory. In HDFS case, this also reduces the number of connections made to DataNodes.

The *ExternalLookup* operator uses a lookup adapter to selectively fetch external records by RID. This operator uses the NC's local snapshot index to retrieve files' metadata. Before opening an external file, the operator contacts the HDFS Name Node to validate the existence of the file. If the file was not found, subsequent RIDs with the same file id are dropped. If the block containing accessed records is found to be local, the operator can skip the connection to the DataNode and read directly from its local disk; otherwise the operator reads the records from one of the HDFS DataNodes that holds the block. The operator parses and sends the fetched records in binary ADM format to the consuming operator in the query execution pipeline.

5.3 Updating an External Dataset

Data can change over time, causing external indexes to become stale. To advance the metadata snapshot associated with an external dataset to the current point in time and update all of its secondary indexes, the *refresh external dataset* AQL statement is used. The refresh operation follows the presumed-abort 2-Phase Commit (2PC) protocol [17] to atomically update an external dataset. We exploit AsterixDB's usage of LSM indexes [6] to perform shadow-based transactional batch updates. An LSM index consists of multiple disk-resident immutable components plus a mutable in-memory component. When beginning an external dataset refresh transaction, a new shadow index component is created. All operations under the refresh transaction will be applied to the shadow component. Such components can handle the addition of new records as well as the deletion of existing records. When a refresh transaction commits, its shadow components are revealed and added to the LSM indexes to be used in answering subsequent queries [17, 6]. Aborting a refresh transaction results simply in the deletion of its shadow components.[2] Our failure handling before the decision to commit is fail-backward and we don't support resume operations.

Each external dataset refresh operation starts by computing a snapshot delta that consists of a set of deleted files, a set of record-appended files, and a set of new files. If all three sets are empty, the operation is complete, and otherwise the delta is recorded on disk and the transaction enters the "presumed abort" state. AsterixDB updates both the snapshot index and the secondary indexes of the dataset undergoing the refresh transaction in each NC. In the shadowed component of the snapshot index, new tuples are added for new files and anti-matter tuples (delete flags) for deleted files. Subsequently, and for each secondary index, a Hyracks job similar to the index building job is constructed to update the index.

[2]Shadow components are only used for external data refresh transactions and are not used with internal datasets.

Deletion of index entries for records in deleted files is important to enhance index performance and to reclaim space, but adding an anti-matter tuple for each deleted record would require expensive scans of deleted index components and yield large shadow components. To avoid this, a special "buddy B+ Tree" is paired with each external LSM index component; it is used to store the file numbers of newly deleted files that potentially had entries in older index components. During an index search, the buddy B+ Trees enable deleted records to be filtered out early. Each buddy B+ Tree is accompanied by a Bloom filter to further reduce the number of I/O operations during searches. When merging LSM index components, the buddy B+ Trees of the index are used to filter out the deleted record entries and reclaim disk space.

A refresh transaction moves into the "ready-to-commit" state when all of the dataset's indexes have their shadow components ready. The CC then records the transaction state on disk and instructs all nodes to commit the transaction locally. After it has committed on all NCs, the delta is added to the ExternalFile dataset and the transaction is marked complete. This use of 2PC enables external indexes to recover from crashes and maintain a consistent state. When bootstrapping, the CC loops over all incomplete transactions to perform global recovery. Transactions in the presumed abort state are rolled back, and transactions in the ready-to-commit state are rolled forward. Since refresh operations always bulk load their changes into shadow components, no record-level logging is required.

5.4 Consistency and Concurrency

The semantics of our logical design forbid external data refresh transactions from affecting external data access in queries that began before the transaction is committed. The semantics are maintained while allowing datasets to be accessed (using indexes or using a scan) and refreshed simultaneously. In this section, we explain how different external datasets' operations correctly interleave.

AsterixDB's CC maintains up to two versions (transiently) of the state of each external dataset plus a pointer to the more recent version. It also tracks the number of queries accessing each version. NCs similarly manage two versions for each external index. Each of these corresponds to one of the versions at the CC and is just a list of pointers to a subset of the index's LSM components. (Index components can be shared between the two versions of the index.)

Before a query accesses an external dataset, the compiler marks it with the id of the most recent version of the dataset and increments the number of queries accessing that version. During query execution, external index search operators use the assigned id to only search the LSM index components that belong to the matching version of the index. When a query finishes execution, the access count for each of its accessed datasets' versions is decremented. When the number of queries accessing the older version of a dataset reaches 0, it becomes *inactive* and can be deleted.

A dataset refresh can start only when the older version of the dataset is inactive. When committing a refresh transaction locally, new versions of the dataset's indexes are created that contain existing index components in addition to newly uncovered shadow components. On a global commit, the pointer to the most recent view of the index is updated and it is then ready to accept queries. This allows queries to run and use external indexes for a dataset that is concurrently being refreshed without affecting their results.

Most operations on external datasets can be executed in parallel. Multiple indexes can be created concurrently. However, refresh operations for a dataset must be serialized. A dataset refresh is also mutually exclusive with index creation, as indexes are created according to a captured snapshot (which is updated through refresh operations).

6. EXPERIMENTAL EVALUATION

We now compare the performance of AsterixDB external datasets, on Hive-created data, against that of Hive itself on the same data files. We also compare external and internal dataset performance for the same data content; this shows the trade-offs involved in moving data into the system versus indexing it externally.

6.1 Experimental Setup

We used a 10 node cluster; each node had a dual core processor, 8GB RAM and 2x 1TB 7200rpm hard drives. AsterixDB (0.8.6) and Hadoop (2.2.0) were set up to utilize all nodes in the cluster. Hadoop was configured with an I/O buffer size of 64 KB, a block size of 64 MB, and a replication factor of 3. We used TPC-H data at scale=250. The data (delimited-text) was put uniformly into HDFS with an equal number of blocks on each of the 20 disks on the cluster. Hive (0.13.0) tables were created (using *insert into select* style queries) with data in three different formats – Text file, Sequence file, and RCFile. Data in the RCFile format used 4 MB as the row group size. Table 1 shows the sizes for the Hive tables in each format. Datatypes were defined in AsterixDB to model TPC-H data. The time and space involved in loading the Lineitem, Order, and Customer data into internal AsterixDB datasets are shown in Table 2. The time includes the cost to fetch and parse data from HDFS, hash-partitioning and sort received records by primary key, and bulk-load them into B-Tree indexes partitioned across the cluster.

The queries in the following section were run in sequence. Each was run multiple times ranging from 5 times for full scan queries to 50 for expensive range lookups and 500 for low cost queries, and results were averaged. Predicate ranges were controlled for each point in the x-axis and their position was selected using a uniform random number generator. In contrast, simply creating an external dataset amounts to an insert of a row in AsterixDB's Metadata and requires negligible space and time.

6.2 Results and Analysis

In this section, we compare the performance of full scan operations using Hive, AsterixDB external datasets, and AsterixDB internal datasets. We also show the time and space implications of indexing different external and internal datasets. Subsequently, we evaluate the performance of indexed access to external versus internal datasets for several different query scenarios.

6.2.1 Aggregation and Lookup Queries (Full Scan)

We first ran queries to aggregate a single attribute and to look up a single record. Aggregates highlight performance when computation is performed on a single attribute, which affects the minimum deserialization requirements (taken advantage of by Hive with its RCFile formatted records). List-

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

Table 1: HDFS raw data files

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

Table 2: Cost of loading data

```
--Aggregate Query
SELECT MAX(l.L_PARTKEY) FROM lineitem l;
--Lookup Query
SELECT * FROM lineitem li
  WHERE li.L_SUPPKEY = a
  AND li.L_LINENUMBER = b
  AND li.L_QUANTITY = c
  AND li.L_EXTENDEDPRICE = d;
--Aggregate Query
SELECT MIN(o.O_TOTALPRICE) FROM orders o;
--Lookup Query
SELECT * FROM orders o
  WHERE o.O_CUSTKEY = a
  AND o.O_TOTALPRICE = b
  AND o.O_OSHIPPRIORITY = c;
--Aggregate Query
SELECT MAX(c.C_BALANCE) FROM customer  c;
--Lookup Query
SELECT * FROM customer c
  WHERE c.C_CNAME = a AND c.C_ADDRESS =
    b;
```

Listing 4: Aggregation and Lookup queries in HiveQL

```
let $litems := for $li in dataset Lineitem
               return $li.partkey
return max($litems);
for $li in dataset Lineitem
  where $li.lsuppkey = a
  and $li.llinenumber = b
  and $li.lquantity = c
  and $li.lextendedprice = d
return $li;
let $price := for $ord in dataset Orders
              return $ord.ototalprice
return min($price);
for $ord in dataset Orders
  where $ord.ocustkey = a
  and $ord.ototalprice = b
  and $ord.oshippriority = c
return $ord;
let $balance := for $cust in dataset Customer
                return $cust.cbalance
return max($balance);
for $cust in dataset Customer
  where $cust.cname = a and $cust.caddress = b
return $cust;
```

Listing 5: Aggregation and Lookup queries in AQL

ings 4 and 5 show the queries in HiveQL and AQL. The parameters a, b, c, and d were picked from specific records in different files. In the absence of an index, each query involved a full scan of the data. Figures 4(a), 4(b) and 4(c) show the results for the Lineitem, Customer and Order datasets, respectively, for data in different platforms and formats. The following are the key observations.

1) ***Querying Internal Data***: Queries against AsterixDB's internal datasets are faster than querying HDFS data, in any format, using either Hive or AsterixDB's external datasets. This is due to internal data being stored in AsterixDB's binary data format (ADM), requiring no translation. Further, the I/O done by AsterixDB is sequential, with a single thread reading from each I/O device with no concurrent I/O activities. For HDFS data, multiple threads may access different splits of data simultaneously and interfere.

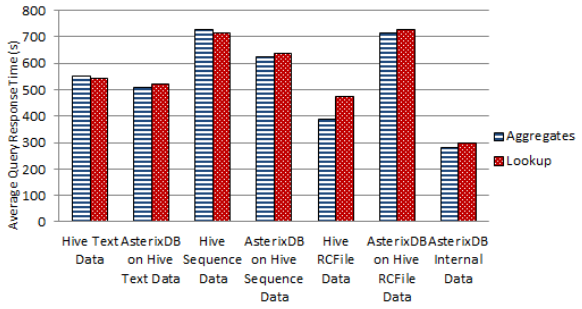
2) ***Querying External Data***

(a) *Text and Sequence File Format*: AQL queries on external data in Text and Sequence File formats in HDFS run faster than the corresponding Hive queries. This is due to the different execution engines of AsterixDB and Hive. An AQL query is compiled into one Hyracks job, while a Hive query is compiled into a Map-Reduce job (or jobs) to run on Hadoop. Hyracks exploits partitioned and pipelined parallelism to offer a more efficient execution model than Hadoop. Additional details on Hyracks can be found in [8].

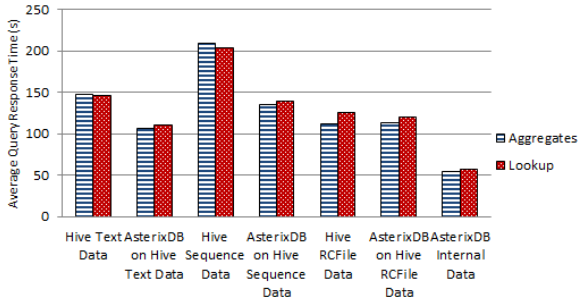
(b) *RCFile Format*: Queries against the largest dataset, Lineitem, for the RCFile format, run faster in Hive than in AsterixDB as an external dataset being queried with AQL. Also, Hive performs relatively worse for simple selects as compared to aggregates. This is due to its need for additional field deserialization. However, this is mitigated by the use of a lazy deserializer that only deserializes fields when needed. AsterixDB, however, parses all the external fields. This difference lessens for datasets with smaller numbers of fields, e.g., Orders and Customers. In addition, AsterixDB does two-step parsing in its current Hive parser, deserializing records into Hive objects before converting them to ADM.

6.2.2 *Joins (Full Scan)*

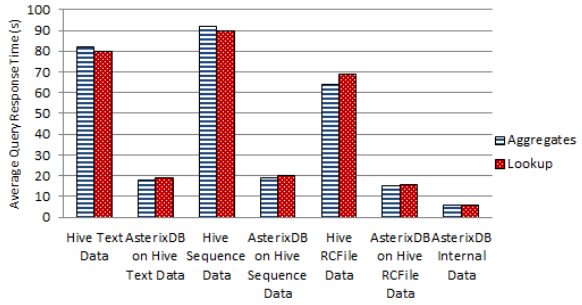
Next, we evaluate a join between Customer and Order data on customer key. The result sizes were limited to around 1000 records via a range predicate on CustKey of Orders. The ranges, a and b, were picked randomly from the range of CustKeys where b is greater than a by 100. Listings 6 and 7 show the join query in HiveQL and AQL respectively. Figure 5 compares the query execution times for different storage platform and data format combinations. AsterixDB employs a completely hash-based join algorithm that is more efficent than the sort-and-shuffle-based join algorithm used by Hive. In addition, Hyracks jobs have lower cost than Map-Reduce jobs. Therefore, joins in AsterixDB run faster than joins in Hive. The internal dataset join achieves the best performance, as only the (selected) Orders records are shuffled to the nodes containing the matching Customer records (since the join key is a primary key) and internal datasets avoid parsing overhead.



(a) Lineitem Dataset



(b) Orders Dataset



(c) Customer Dataset

Figure 4: Performance comparison: Aggregate and Lookup queries (Full Scan)

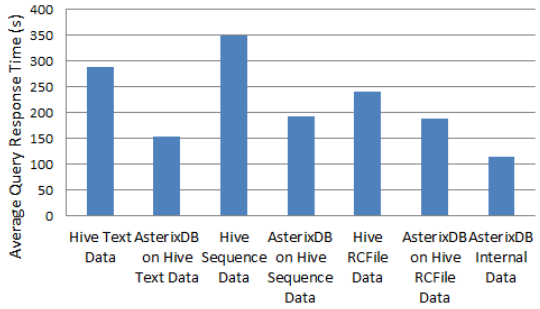


Figure 5: Performance comparison: Join of Customers and Orders (Full Scan)

6.2.3 *Constructing Indexes: Internal vs. External*

Next, we explore the costs of building secondary indexes on internal AsterixDB data versus external HDFS data. As shown in Table 3, indexing external data is slower than internal data. To see why, recall the index-construction pipeline from Figure 2. Indexing an external dataset requires parsing the data into ADM format. In addition, larger composite

TPC-H Data	Key	Format	Time (secs)	Size (GB)
Lineitem	order key	Text	1458	48.8
		Sequence	1678	48.8
		RCFile	1796	58.6
	part key	Internal	1313	40
		Text	1632	48.8
		Sequence	1802	48.8
		RCFile	2156	58.6
Orders	customer key	Internal	297	7.7
		Text	356	12
		Sequence	422	12
		RCFile	473	14.5
	order key	Text	300	48.8
		Sequence	362	48.8
		RCFile	341	58.6
Customer	customer key	Text	38	1
		Sequence	39	1
		RCFile	49	1.47

Table 3: Indexing of external and internal datasets

```
SELECT cust.ccustkey, cust.cname, ord.oorderkey
    FROM orders ord JOIN
customer cust ON(ord.ocustkey = cust.ccustkey)
WHERE ord.ocustkey > a and ord.ocustkey < b;
```

Listing 6: Join query example in HiveQL

```
for $ord in dataset Orders
for $cus in dataset Customer
where $ord.ocustkey = $cus.ccustkey
and $ord.ocustkey > a and $ord.ocustkey < b
return { "key":$cus.ccustkey,
   "name":$cus.ccname,
   "orderkey":$ord.oorderkey };
```

Listing 7: Join query example in AQL

```
count(for $ord in dataset Orders
      where $ord.oorderkey > a and $ord.oorderkey < b
      return $ord);
```

Listing 8: Range query on Orders using order index

Record IDs slow down the sort operation. Further, an HDFS record reader may scan multiple files from partitions on the same I/O device, causing interference and non-deterministic delays. (The Hadoop 2.2.0 API doesn't provide a way to determine the I/O device hosting each block and to schedule accordingly.) Note that the index size for the RCFile format is larger due to the additional row number field in each RID.

6.2.4 *Index Access: Internal vs. External*

Having built indexes on both internal and external datasets, we then ran queries to use them. Each query was run for multiple data ranges to show its performance for different result sizes. The parameters a and b in Listings 8, 9, 10, and 11 were substituted with values picked from their respective data ranges using a uniformly distributed random value generator. The difference between a and b was used to control the result sizes. Throughout our experiments, the measured queries just count the results rather than return-

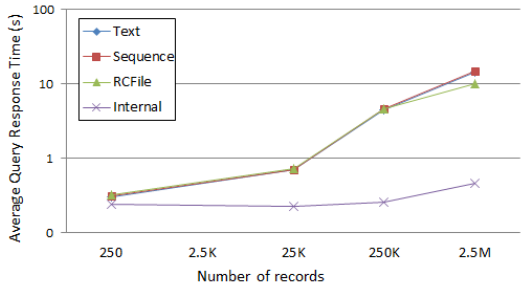


Figure 6: AsterixDB range queries on Orders dataset using order index

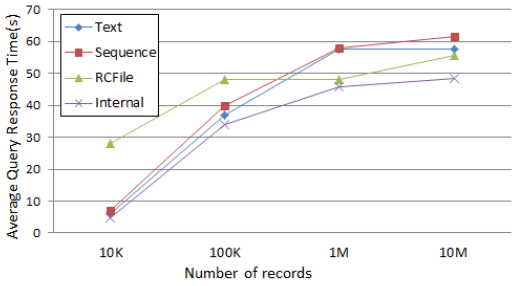


Figure 7: AsterixDB range queries on Orders dataset using customer index

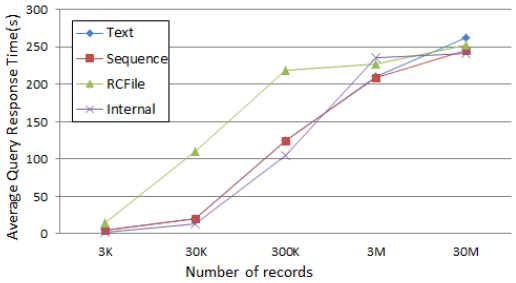


Figure 8: AsterixDB range queries on Lineitem dataset using part index

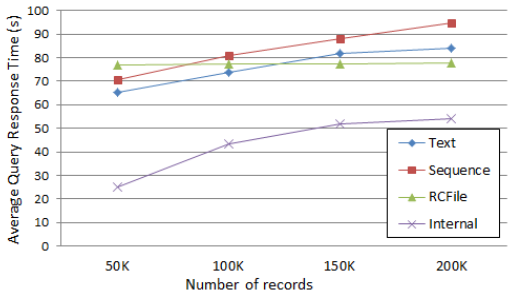


Figure 9: AsterixDB index-based join of Orders and Customers datasets on customer key

ing their data; we did so to isolate computation costs from the cost of result delivery. We now examine the results.

(1)*External Secondary Index vs. Internal Primary Index:* Queries on primary keys benefit when data storage is internal using a primary index. To gain similar efficiency, such queries on external data need secondary indexes (as external datasets lack primary keys). We ran an example query (Listing 8) on an internal dataset as well as on external datasets with different HDFS formats. Figure 6 shows the results. At a result size of 250 records, internal and external data perform similarly; times are dominated by query compilation and running a small Hyracks job. As we increase the result size to 2.5 million records by altering the order key predicate, the time for the internal dataset remains well below 1 second. In contrast, the time for external datasets increases by factors of 10 and 14 for the RCFile and Sequence/Text formats. Querying an internal dataset benefits from evenly distributed processing due to hash-partitioning of records on search key. Moreover, the primary index access requires a sequential scan just of the index leaves, as opposed to random accesses of leaves and internal nodes of primary indexes when accessing data via secondary indexes. Querying external datasets involves additional steps (and overheads) including a secondary index search, sorting RIDs to ensure more clustered access, using RIDs to read external records, and parsing records into ADM binary format.

The generated raw data for the Orders dataset was initially sorted on the order key. Hence, external index access in the query involved sequential I/O. In the case of the RCFile format, groups of records (4 MB in size) were read together, whereas records in the Text/Sequence file formats were read one at a time (causing additional context switching overhead). For this reason, querying external data in the RCFile format was observed to be faster than querying the same data in the Sequence and Text formats (by a factor of 1.4) when reading 2.5 million records.

(2)*Secondary Index Access - Internal vs. External:* To compare secondary index access on external and internal datasets, we ran the range queries shown in Listings 9 and 10. Such a range query does not benefit from hash-partitioned distribution of data records. Furthermore, secondary access paths for internal datasets incur additional CPU and I/O costs to search the primary index to fetch the records. External datasets, on the other hand, can directly use the records' physical locations (embedded in their RIDs) to fetch them from HDFS. Figures 7 and 8 show the results with different range predicates. Both figures show fairly comparable performance for internal data and external data with Text and Sequence formats. In contrast, secondary index access for the RCFile format starts out being about 3 times slower than the other formats (at 10K in Figure 7 and at 30K in Figure 8). This is because accessing RCFile records requires reading their containing row groups (4 MB in size). Starting at the 100K range for the Orders dataset and 300K for the Lineitem dataset, the rate of increase of the RCFile access time decreases, and the cost eventually becomes lower than the other formats. This is due to the increase in the ratio of records to row groups, as the benefits of performing sequential reads and less context switching eventually outweighs the cost of reading extra records. The I/O and parsing costs associated with reading Text and Sequence records are similar. A single read etches a minimum of 64 KB for both formats. However, the actual location of their HDFS blocks and indexes on disks are different, creating small differences in their performance for individual queries.

(3)*Index Nested Loop Joins - Internal vs. External:* Finally, we evaluate an AQL index nested loop (INL) join (Listing 11). This query counts the output of a select on the

```
count(for $ord in dataset Orders
      where $ord.ocustkey > a and $ord.ocustkey < b
      return $ord);
```

Listing 9: Range query on Orders using customer index

```
count(for $li in dataset Lineitem
      where $li.lpartkey > a and $li.lpartkey < b
      return $li);
```

Listing 10: Range query on Lineitem using part index

```
count(for $ord in dataset Orders
for $cus in dataset Customer
where $ord.ocustkey = /*+indexnl*/ $cus.ccustkey
and $ord.ocustkey > a and $ord.ocustkey < b
return $cus);
```

Listing 11: Index nested loop join query

Orders dataset using a predicate on the customer key followed by an INL join on this key with the Customer dataset. As shown in Figure 9, the join runs faster against internal datasets by a factor ranging between 1.4 and 3. For the internal dataset, only Orders records are re-partitioned since the nodes with matching Customer records are known (Customer is hash-partitioned on the customer key). This is not so for external datasets, where partitioning knowledge is unavailable and each produced record must be *broadcast* to all participating nodes. To get the matching record, the internal dataset need only search the primary index, while external datasets first search their secondary indexes and then perform seek, read and parse operations before joining the records. Note that join performance is fairly similar across the three external file formats. As seen previously, RCFile access is slower than the other formats at smaller ranges due to the additional I/O cost. RCFile access cost remained fairly flat even when the join size increased from 50K records to 200K records. This shows that reading more records when accessing RCFile data doesn't necessarily increase the I/O cost, which dominates for this query. The text and Sequence formats showed linear increases in access time while maintaining their difference (around 35 seconds) from internal joins.

7. CONCLUSION

We have described how AsterixDB provides efficient query access to data living outside the system (e.g., in HDFS). Pluggable adapters and parsers provide an extensible framework to support a variety of non-native data sources and formats. To support queries with small to medium selectivity, we added incrementally refreshable distributed indexes that provide carefully enforced access semantics to ensure answer consistency across access paths. We also explained how consistency is ensured when building, using, and refreshing external indexes as well as how failures are handled. We showed that AsterixDB's support for external HDFS data provides both good full scan performance (compared to Hive) as well as significant improvements from indexing. We also compared external and internal data access costs for different queries to gauge the relative performance of external data access. AsterixDB will release these features in its first official Apache incubation release in 2015. Applying these techniques on new data formats (e.g., Parquet or ORC) and/or measuring performance against new "SQL on Hadoop" platforms (such as Impala, Stinger, or SparkSQL) are possible future work. We may also explore HDFS's new inotify feature to automate external index refresh and investigate support for resuming index refresh on failures.

Acknowledgements This work was supported by NSF IIS award 0910989 and CNS awards 1305430 and 1059436.

8. REFERENCES

[1] Apache Hadoop. http://www.hadoop.org/.
[2] Apache Hive. http://oozie.apache.org.
[3] A. Abouzied et al. Invisible Loading: Access-driven Data Transfer from Raw Files into Database Systems. *Proc. EDBT Conf.*, 2013.
[4] I. Alagiannis et al. NoDB: Efficient Query Execution on Raw Data Files. *Proc. SIGMOD Conf.*, 2012.
[5] S. Alsubaiee et al. AsterixDB: A Scalable, Open Source BDMS. *Proc. VLDB Endow.*, 7(14), 2014.
[6] S. Alsubaiee et al. Storage Management in AsterixDB. *Proc. VLDB Endow.*, 7(10), 2014.
[7] S. Blanas et al. Parallel Data Analysis Directly on Scientific File Formats. *Proc. SIGMOD Conf.*, 2014.
[8] V. Borkar, M. Carey, et al. Hyracks: A Flexible and Extensible Foundation for Data-intensive Computing. *Proc. ICDE Conf.*, 2011.
[9] J. B. Buck et al. SciHadoop: Array-based Query Processing in Hadoop. *Proc. ACM Int'l. Conf. on High Perf. Comp., Netw., Storage and Analysis*, 2011.
[10] Y. Cheng and F. Rusu. Parallel In-situ Data Processing with Speculative Loading. *Proc. SIGMOD Conf.*, 2014.
[11] J. Dittrich et al. Hadoop++: Making a Yellow Elephant Run Like a Cheetah (Without It Even Noticing). *Proc. VLDB Endow.*, 3(1-2), 2010.
[12] M. Y. Eltabakh, F. Özcan, Y. Sismanis, et al. Eagle-eyed Elephant: Split-oriented Indexing in Hadoop. *Proc. EDBT Conf.*, 2013.
[13] V. R. Gankidi et al. Indexing HDFS Data in PDW: Splitting the Data from the Index. *Proc. VLDB Endow.*, 7(13), 2014.
[14] S. Idreos et al. Here Are My Data Files. Here Are My Queries. Where Are My Results? *Proc. CIDR Conf.*, 2011.
[15] K. Lorincz, K. Redwine, and J. Tov. Grep versus FlatSQL versus MySQL: Queries using UNIX tools vs. a DBMS, 2003.
[16] J. Melton et al. SQL and Management of External Data. *ACM SIGMOD Rec.*, 30(1), 2001.
[17] C. Mohan et al. Transaction Management in the R* Distributed Database Management System. *ACM TODS*, 11(4), 1986.
[18] Y. Xu, P. Kostamaa, and L. Gao. Integrating Hadoop and Parallel DBMS. *Proc. SIGMOD Conf.*, 2010.