A Performance Study of AsterixDB

Keren Ouaknine
School of Engineering and Computer Science
Hebrew University of Jerusalem, Israel
Email: keren.ouaknine@mail.huji.ac.il

Michael Carey
Bren School of Information and Computer Sciences
University of California, Irvine
Email: mjcarey@ics.uci.edu

***Abstract*— Apache AsterixDB is a relatively new Big Data management platform providing ingestion, storage, management, indexing, querying, and analyses of vast quantities of semi-structured information on scalable computer clusters. This paper compares the execution and performance of an early release of Apache AsterixDB with two popular platforms, Apache Hadoop and HPCC Systems, over the 17 PigMix benchmark query scenarios. We discuss the results and also how they have influenced the AsterixDB effort.**

***Keywords*-AsterixDB; Hadoop; HPCC; Performance;**

I. Introduction

Social networks, online communities, mobile devices and instant messaging applications generate complex, unstructured data at a hight rate, resulting in large volumes of data. This creates challenging scenarios for data management systems aiming to ingest, store, index and analyze such data efficiently.

Apache AsterixDB [1], [2] is a recent open source Big Data platform co-developed at UC Irvine, UC Riverside and UC San Diego. It is able to ingest, manage, index, query and analyze mass quantities of semi-structured data. Based on ideas from parallel databases and first generation Big Data platforms, AsterixDB is a next-generation open-source platform running on large, shared-nothing, commodity computing clusters. To explore its potential benefits, in 2015 we decided to analyze and compare the performance of AsterixDB to two popular Big Data platforms using the PigMix benchmark running on all the 17 PigMix use-cases.

II. Study Context

A. Apache AsterixDB

AsterixDB is a parallel, semistructured information management platform that provides the ability to ingest, store, index, query, and analyze mass quantities of data. It has a flexible data model (ADM) that is a superset of JSON and a query language (AQL) comparable to languages such as Pig [3], and Hive [4]. Through ADM and AQL, AsterixDB supports native storage and indexing of data as well as access to external data (e.g. data in HDFS). AsterixDB uses the Hyracks data-parallel platform [5] as its runtime engine.

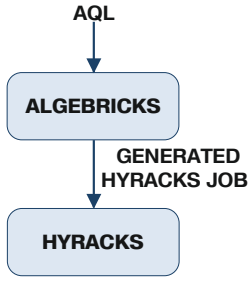


Figure 1: Compilation of AsterixDB queries

B. Algebricks

To process a query, Apache AsterixDB compiles an AQL query into an Algebricks [5] algebraic program also known as the logical plan as illustrated in Fig. 1. This plan is then optimized via rewrite rules, as described in Appendix A, that reorder the Algebricks operators and introduce partitioned parallelism for scalable execution. After the optimization step, a code generation step translates the resulting physical query plan into a corresponding Hyracks Job that uses the Hyracks engine to compute the requested query results. Finally, the runtime plan is distributed on the system and executed locally on every slave of the cluster.

The AsterixDB query optimizer takes into consideration many properties of the data such as the data partitioning and ordering and decides according to a set of rules (which are the core of Algebricks) what the steps should be to execute the query in the most performant manner.

C. PigMix

PigMix [6] was developed by Hortonworks in 2009 to test and track the performance of the Pig query processor from version to version. It has been used in studies ranging from cloud computing [7] to user engagement modeling [8] and security [9]. It consists of 17 queries operating on 8 tables, and tests a variety of operators and features such as sorts, aggregations, and joins. In Table I, we describe the eight input tables and their respective schemas representing Yahoo! users requesting for web pages. The main table, called page_views, contains fields such as the query terms that brought the user to the page as well as the time that the user spent on that page. Table II describes the 17 queries.

978-1-5386-2715-0/17/$31.00 ©2017 IEEE

Table I: The 8 input datasets of PigMix

Name of table	Schema	Purpose
page_views	user name, action, timespent, query_term, ip_addr, timestamp, estimated_revenue, page_info, page_links	main table
users	name, phone, address, city, state, zip	user information
page_views_sorted	same as page_views	pre-requisite table for merge-join evaluation
users_sorted	same as users	pre-requisite table for merge-join evaluation
power_users	name, phone, address, city, state, zip	small table to test replicate join
widerow	name and 499 integers	very wide row table
widegroupbydata	same schema as page_views repeated three times	aggregation of wide keys

Table II: Description of each query in the PigMix benchmark

Q1	list of users with a count of the pages they viewed or the number of links matching a specific value ('b') depending on the user's action
Q2	the total revenue for each user present in both page_views and power_users
Q3	the total revenue for each user present in both page_views and power_users
Q4	the number of actions (page_views) per user
Q5	all the users from page_views that do not have a matching user from the table users
Q6	groups the page_ views dataset by user, query term, ip address, and time. It returns the cumulated time spent for each group.
Q7	count of the morning and afternoon accesses for each user
Q8	the total time spent and estimated revenue for the page_views dataset
Q9	the page views records ordered by query term
Q10	the page views records ordered by a combination of query term, estimated revenue, and time spent
Q11	returns records of page views and widerow distincted by username
Q12	returns (i) the anonymous page_view records grouped by query term and sums the time spent for each group, (ii) the non-anonymous page_view records with a query term not null grouped by user and sums the estimated revenue for each group(iii) the non-anonymous queries with a null query term, grouped by user, and counts the action for each group
Q13	user names in page views with their phones from the users table
Q14	user names in page views with their users from the users table
Q15	a count of the action, a sum of the revenue, and an average of the time spent. The count includes unique actions only. The sum adds unique estimated revenue. The average includes distinct time spent only. The query groups by one field, removes duplicates within groups, and aggregates (count, sum, average) each group
Q16	the sum of the estimated revenue for each user
Q17	the users grouped by all their fields

1) Data generation: The actual data of PigMix is randomly generated. One can vary the amount of data to generate by specifying the number of page_views rows. Our input sizes started at 10 million rows, i.e 16 GB, which is the scale at which Hortonworks experiments with the benchmark. We then scaled page_views up to 270 million rows, which is 432 GB, partitioned over ten nodes to test the platforms at a Big Data scale.

The PigMix input table users has unique keys derived from page_views and additional columns such as phone, address, etc. The input table *power_users* has the same schema as *users* but is smaller. It has 500 rows generated by skimming unique names from users. This produces a table that can be used to test replicated joins where one table is typically small. The input table *widerow* has 500 fields consisting of one string and 499 integers. The PigMix benchmark also has two sorted tables, *page_views_sorted*, and *users_sorted*, to be used in a merge-join query where the two inputs to the join need to be sorted prior to the join execution. Additionally, the benchmark uses a sample input table which retains only half of the records in power_users and uses the table in another join query. Lastly, PigMix also has an input table *widegroupbydata* with the fields of *page_views* replicated three times to cover the case of a wide group key. Full details on all of these tables (including field lengths, etc) can be found on the PigMix webpage [6].

III. Performance Results and Analysis

The PigMix benchmark [6] has four main query types: sort, join, aggregate, and union. In this section, we analyze the execution plans and execution times of these types of queries across the four systems circa 2015: AsterixDB 0.8.7, Pig 0.13.1, Hadoop 1.2.1, and HPCC community 5.0.2-1 as summarized in Table III.

Hardware: We used a cluster of 10 dual-socket, dual-core AMD Opteron servers. Each server has 8 GB of memory and dual 1 TB 7200 RPM SATA disks. The servers were all connected via 1 GB Ethernet on the same switch. Note that despite running on the same hardware and having the same input data, each system has a different representation of that input data. For example, the stored size of the main input table *page_views* is 436 GB on Hadoop, 376 GB on HPCC Systems, and 585 GB on AsterixDB.

Table III: Query performance for our four scenarios (sec)

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17
PIG	2,427	856	1,236	1,033	1,027	1,251	1,221	1,026	7,304	9,577	1,041	1,222	1,118	978	1,277	1,201	4,678
MapReduce	2,403	973	1,248	1,099	1,180	1,158	848	1,153	6,580	8,475	1,239	3,394	1,153	1,164	1,055	1,144	4,725
HPCC	1,382	1,189	1,292	1,286	1,409	1,377	1,372	1,223	29,687	29,627	1,168	3,509	1,302	1,396	3,894	1,277	1,400
AsterixDB	29,207	15	1,383	1,175	1,738	1,902	9,668	1,180	20,863	20,710	1,203	2,514	1,549	1,006	3,901	1,454	4,831

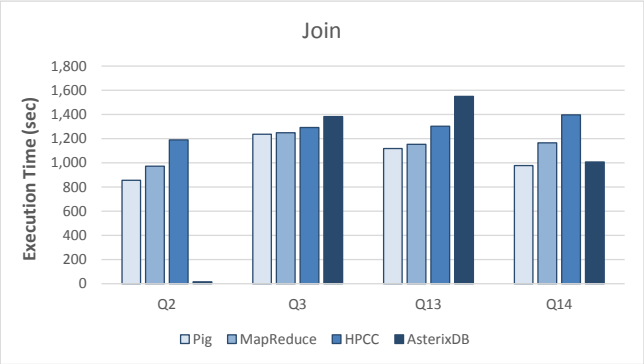


Figure 2: Performance results for the join queries

A. Join Queries

We begin our analysis with Join, which is used very frequently in database systems. Pig implements three types of joins: a replicated (also known as a broadcast) join, a merge join, and a skew join. These are benchmarked in four of the PigMix queries using hints and are described below:

(i) A replicated join has one of its two inputs typically small. The small table is broadcast entirely to all nodes and joined at each one with a portion of the large table. A major advantage of this join is that the large dataset does not need to be repartitioned to be joined with the small dataset. In Pig and HPCC the input is broadcast. A match is looked for in the small table for each record in the large table using a nested-loop join. Given an appropriate join hint, AsterixDB executes the join using an index-nested loops join, so its execution time is much faster than the other systems as shown for Q2 on Fig. 2.

(ii) In a skew join particular to Pig, one dataset is partitioned and the other one is streamed. This join has two phases: (1) First comes a preparation phase, during which Pig takes samples to estimate the key distribution. In order to reduce spillage, the sampler conservatively estimates the number of rows that can be sent to a single reducer based on the memory available for the reducer. The partitioner uses the key distribution to send the data to the reducer in a round robin fashion. The mapper task uses the distribution file to copy the data of the other table to each of the reduce partitions matching the key. Since more than one reducer can be associated with a key, the table records that match the key need to be copied over to each of those reducers. (2) The tuples from the two tables are then joined.

(iii) A third type of join intends to test the case where the data is already sorted. Pig and HPCC have a merge join as an available join strategy, whereas AsterixDB does not so instead it uses a hash join to execute the join.

1) Results Analysis for Join: Fig. 2 shows all the join query performance results. The results are fairly similar except for the superiority of AsterixDB in Q2, which was due to indexes as explained above. As for the other queries, where no index was used, AsterixDB was slightly slower than Pig which is not surprising as it has a larger input size.

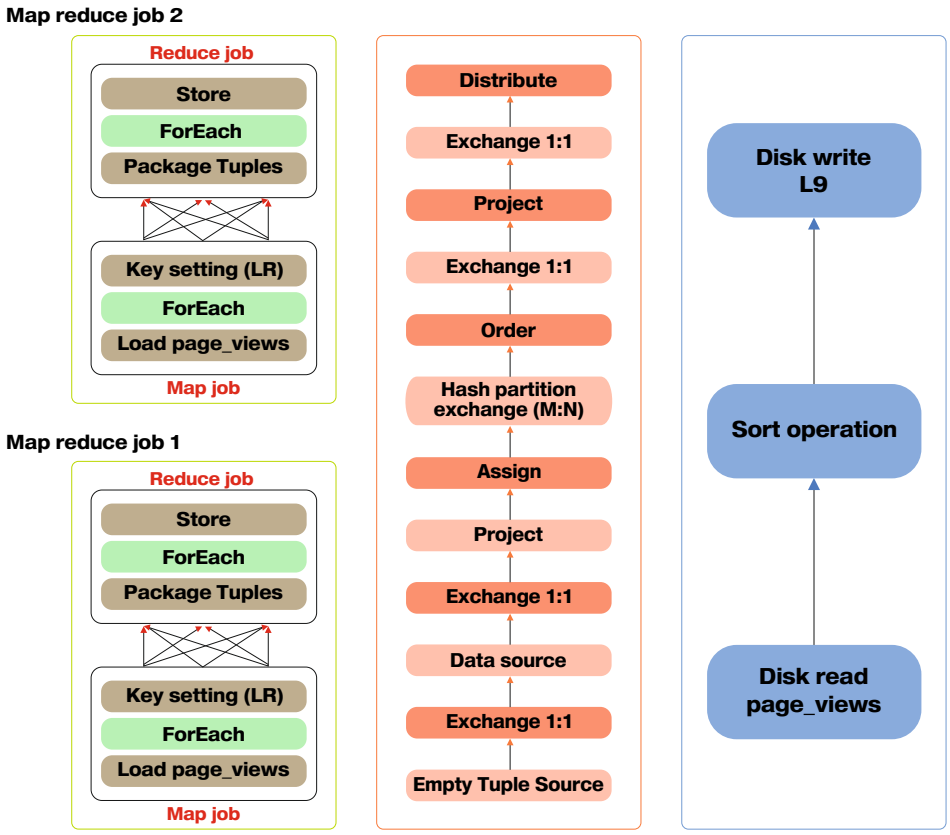
The three execution plans for a join (with aggregation) on large input tables (Q3) are shown in Fig. 5 for each of the systems, and the plans are similar.

B. Sort Queries

The sort operator is a widely used and is a relatively expensive operation in parallel processing systems [10]. Sort is covered in the PigMix benchmark by two queries (Q9, Q10) and involves three steps: determining a vector of splitting ranges, repartitioning the input by these ranges, and sorting the resulting partitions locally using an external merge sort. In the first step, the systems determine the ranges manually or using automatic sampling techniques. In the second step, all systems repartition the data according to the ranges. In the third and last step, the data is locally sorted on each node.

1) Sampling techniques for sort optimization: The first step of sort determines ranges so that all the data going to one node is "pre-sorted" compared to that of another node. The sampling step provides an estimate for the key distribution to plan a balanced repartition of the data between the nodes. Each system has its own sampling and redistribution technique. The Pig optimizer runs a MapReduce job to sample the data, then the partitioner redistributes the data according to these ranges. This step happens between the map and the reduce phase i.e., the shuffle phase. The data sent by the mappers to the reducers is sorted using the Hadoop framework. Since the data is range-partitioned and locally sorted, the data is globally sorted. Note that to minimize skew, Pig can (unlike HPCC and AsterixDB) distribute records with the same key to two different partitions. This has a positive impact on the performance of Pig for sort queries with skewed data. On HPCC, ranges are also determined automatically. In AsterixDB[1] and hand-written MapReduce Java code, the ranges are defined by the user manually.

[1]Automatic sampling is currently under development in AsterixDB.



(a) Pig/Hadoop plan (b) AsterixDB plan (c) HPCC System plan

Figure 3: Execution plan of a SORT query

2) The Execution Plans and Result Analysis: The execution plan for a sort query (Q9, Q10) on all three platforms is shown in Fig. 3. On the left hand side of the figure is the Pig plan. It has two MapReduce jobs concatenated, one for sampling and the second one for sorting. The first MapReduce job collects samples and builds ranges according to which the following MapReduce job will repartition the data. The data is then sorted by the MapReduce framework transparently. The middle graph shows the execution of AsterixDB. It is a sequence of Hyracks runtime operations, mainly scan, project, sort and merge. Note that AsterixDB is the only system to use an external merge sort rather than quicksort for the local sort. The HPCC plan doesn't provide us much details; however we noticed a high skew of the data during execution which slows down the entire job. In Fig. 4, we can see a significant shorter execution time for Pig and MapReduce as compared to the AsterixDB and HPCC Systems. This is due to the fact that Pig can handle skew involving a single key value, e.g. many null values, whereas AsterixDB and HPCC don't have this capability. MapReduce also shows a short execution time due to a similar fine tuning of the partitions.

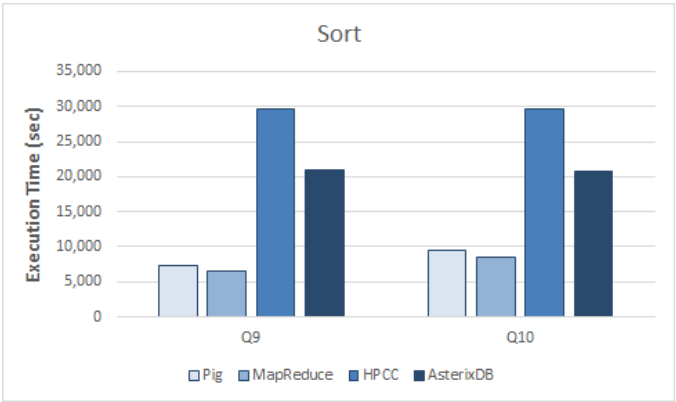
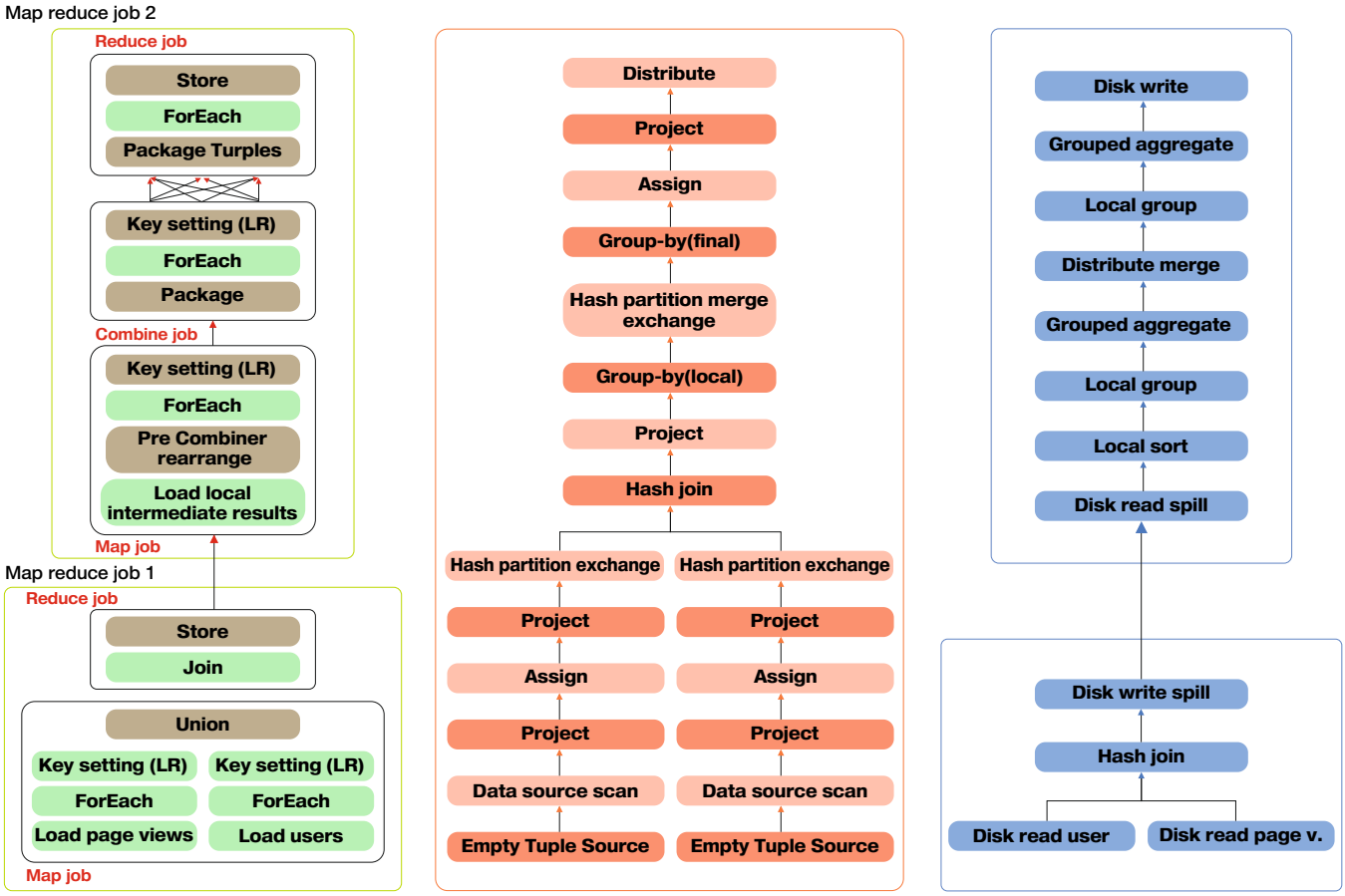


Figure 4: Performance results for the sort query (Q9)

C. Group-By Queries

The group-by operator groups tuples by one or more columns. In the PigMix benchmark, ten of the seventeen queries execute grouped aggregations. There are two common types of group-by implementations: (i) Sort-based group-by, which locally sorts and groups the partitions; this step provides a partial aggregation. The intermediate results are



(a) Pig/Hadoop plan (b) AsterixDB plan (c) HPCC System plan

Figure 5: Execution plan of a JOIN query + aggregation

then shuffled according to the group-by key(s). Following that, the final aggregation phase executes. If there is an index on the group-by key, the local sort phase can be eliminated from the plan. (ii) Hash aggregation, which builds a hash table of the aggregates for the groups. Each record updates a current group in the table or creates a new group.

AsterixDB, HPCC, Pig and MapReduce all execute their aggregations using a sort-aggregate by default as described in (i).

1) Results Analysis for Aggregation: The group-by results are shown in Fig. 6 and Fig. 7. In the queries Q1 and Q7 in Fig. 6, there is slow execution on AsterixDB due to a materialization issue (an unnecessary persistence step of intermediate data). For all other aggregation queries in Fig. 7, we can see similar timings for all systems except in Q12, Q15, and Q17. Q12 is actually three queries that could be factorized, but this is only done in Pig. There is a similar issue in Q15. Q17 operates on a table which was tripled horizontally (schema wise) compared to the main table, so the difference between the systems is significantly higher. The execution plan for a basic aggregation query (Q4) on all three platforms is shown in Fig. 8.

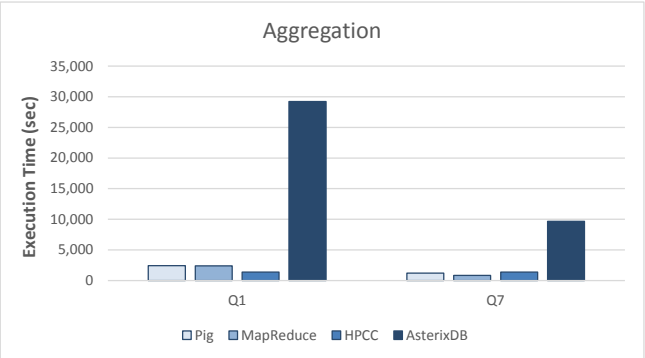
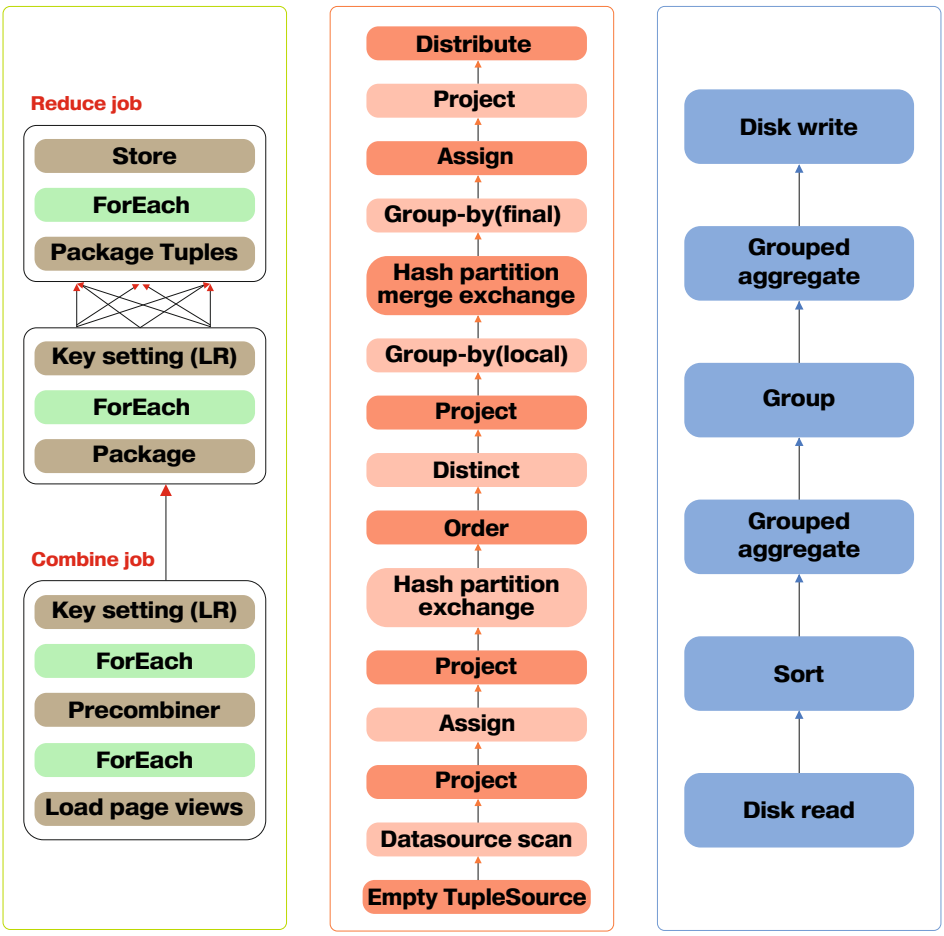


Figure 6: Performance results for Aggregations Q1, Q7

Figure 7: Performance results for Aggregations



(a) Pig/Hadoop plan (b) AsterixDB plan (c) HPCC System plan

Figure 8: Execution plan for aggregation query

D. Differences

Babu and Herodotou [11] have offered a list of criteria to use to compare massively parallel databases. We use some of these criteria in Table IV to highlight the key conceptual differences between Hadoop (and Pig), HPCC and AsterixDB.

The design of Hadoop is oriented towards providing high reliability to long jobs at the tradeoff of raw performance. Intermediate results are written to disk between each mapper and reducer and to HDFS between one MapReduce job and another in a multi-job query. Therefore, when several MapReduce jobs are chained to execute a query, we can expect a slowdown in the execution. This allows MapReduce jobs to tolerate failures within the execution of a job with minimal computing loss. AsterixDB and HPCC do not have this capability and therefore will lose the computing progress of a failed job. Another feature of Hadoop that speeds up jobs is called speculation. Speculation runs “forks” of the same task on several nodes and returns the first one to complete. This can shorten long execution tails due to a slow machine (but not due to skew). Also, Hadoop can reschedule a task on another machine transparently.

IV. Contributions to AsterixDB

Unlike the other systems tested, AsterixDB is a "younger" system. Therefore many features needed to be implemented or improved in order to import the PigMix queries and run them all successfully. As such, we opened and solved many issues as detailed in Table V. Among them, were plan optimization issues, auto-generation problems, AQL issues, aggregation and selection issues, etc. We implemented the union operator on AsterixDB. We added a new result delivery mode (known as *synchronous deferred*). Finally, we also built a graph visualization tool. This work was thus very helpful to the current AsterixDB hardening effort. More details of these issues is on the AsterixDB JIRA [12].

Table IV: Conceptual differences between Hadoop, HPCC and AsterixDB.

Parameter	Hadoop (and Pig)	HPCC	AsterixDB
Processing paradigm	MapReduce programming model	Flexible dataflow	Flexible dataflow
Task uniformity	Same map/reduce task on all nodes	Tasks can vary from node to node	Tasks can vary from node to node
Internode communication	Never directly (always file based)	Sometimes	Sometimes
Repartitioning	Between mapper and reducer only	At any stage	At any stage
Reliability	Configurable through HDFS	One or two replicas	One to three replicas
Fault tolerance	High granularity at the task level	Entire job fails unless predefined checkpoints	Entire job fails
Schema	Schema-less (unstructured data)	Schema required	Schema optional

Table V: Contributions to AsterixDB

Feature/Bug	Purpose	Status
Sorting (#902)	sort on primary key	resolved
Group by(#899)	group-by on nullable field	resolved
Type creation (#893)	Transaction flush failure	resolved
Aggregation (#970)	Performance improvement	resolved
Data insertion (#968)	Performance improvement (days to minutes)	resolved
Materialization during loading (#1039)	WAF Files larger than the src input were generated (!)	won't fix
Aggregation error (#949)	count operator	resolved
Storage (#948)	Read and store data	resolved
Left outer join(#903)	on a nullable field did not work	resolved
Count (#898)	Make it work on large sets (ArrayIndexOutOfBound)	resolved
(#888)	cannot sort on primary key	resolved
ADM (#897)	ADM file scan with certain characters	resolved
Auto-generated field (#889)	Field auto-generated isn't generated	resolved
Type definition (#885)	Large type definition failed	resolved
String value (#867)	Casting issue	resolved
Type computing (#865)	Records not generated	resolved
Missing bytes (#3749)	Data gets lost	resolved
Complex type (#904)	Type with more than one list cannot be created	resolved
Complex type (#972)	bag of list of records cannot be created	resolved
Graph visualizer	Build a graph visualizer to represent the operators flow in Algebricks	completed

V. Conclusions and Future Work

We have reported on a comparison of the 2015 performance and execution of AsterixDB with Hadoop MapReduce, Pig and HPCC Systems. We used the PigMix benchmark, which covers various scenarios, including: sort, join, and several aggregations. We ported the PigMix queries to AQL and added many contributions to AsterixDB to run PigMix on AsterixDB successfully. Pig performed best on join queries due to its ability to work well with skewed data (e.g. many null values). However, AsterixDB returned significantly faster when an index existed on the join fields. Similarly, Pig and MapReduce outperformed HPCC and AsterixDB on sort queries thanks to efficient sampling and a resulting ability to manage value skew well and a faster local sort algorithm. Lastly, aggregation queries were mostly faster on HPCC since some of the queries benefited from its non-materialization approach.

All of the systems tested have evolved since these experiments were conducted. Additionally, new systems have appeared on the scene, such as Spark. As future work, it would be interesting to rerun these experiments again on the latest versions of the broader set of available systems.

Appendix A. The Algebricks Rules

Table VI below contains the list of Algebricks rules.

Appendix B. Data Type Definitions

The statements below illustrate the PigMix data type and dataset definitions in AsterixDB.

```
1
2 create dataverse pigmix;
3
4 create type page_info_type as open {}
5
6 create type page_views_type as closed {
7 user: string,
8 action: int32,
9 timespent: int32,
10 query_term: string,
11 ip_addr: int32,
12 timestamp: int32,
13 estimated_revenue: double,
14 page_info: page_info_type,
15 page_links: {{ page_info_type}}
16 }
17
```

```
create type power_users_type as closed {
id: int32,
name: string?,
phone: string?,
address: string?,
city: string?,
state: string?,
zip: int32?
}

create type widegroupbydata as closed {
user: string,
action: int32,
timespent: int32,
query_term: string,
ip_addr: int32,
timestamp: int32,
estimated_revenue: double,
page_info: page_info_type,
page_links: {{ page_info_type}},
user_1: string,
action_1: int32,
timespent_1: int32,
query_term_1: string,
ip_addr_1: int32,
timestamp_1: int32,
estimated_revenue_1: double,
page_info_1: page_info_type,
page_links_1: {{ page_info_type}}
}

create dataset page_views(page_views_type)
primary key user;

load dataset page_views using localfs
(("path"="localhost:///page_views_test.adm"),("format"="adm"));
```


ACKNOWLEDGMENTS

This effort has been partially supported by NSF CNS award 1305430.

REFERENCES

[1] "AsterixDB Apache website," http://asterixdb.incubator.apache.org/.

[2] S. Alsubaiee, Y. Altowim, H. Altwaijry, A. Behm, V. Borkar, Y. Bu, M. Carey, I. Cetindil, M. Cheelangi, K. Faraaz *et al.*, "AsterixDB: A Scalable, Open Source BDMS," *Proceedings of the VLDB Endowment*, 2014.

[3] "Overview of Pig Apache," http://pig.apache.org/docs/r0.9.1/api/overview-summary.html.

[4] "Hive," https://hive.apache.org.

[5] V. Borkar, Y. Bu, E. P. Carman Jr, N. Onose, T. Westmann, P. Pirzadeh, M. J. Carey, and V. J. Tsotras, "Algebricks: A data model-agnostic compiler backend for Big Data languages," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*. ACM, 2015, pp. 422–433.

[6] "Apache Pigmix website," http://cwiki.apache.org/confluence/display/PIG/PigMix.

[7] J. Ortiz, V. De Almeida, and M. Balazinska, "A Vision for Personalized Service Level Agreements in the Cloud," in *2nd Workshop on Data Analytics in the Cloud*. ACM, 2013.

[8] M. Lalmas, "User Engagement in the Digital World," *Disponible en*, 2012.

[9] James Stephen, et al, "Program Analysis for Secure Big Data Processing," in *ACM/IEEE Conf. on Automated Software Eng.*, 2014.

[10] R. Ramakrishnan and J. Gehrke, "Database management systems," 2000.

[11] S. Babu and H. Herodotou, "Massively Parallel Databases and MapReduce Systems," *Foundations and Trends in Databases*, 2013.

[12] "Apache AsterixDB JIRA Issues," http://issues.apache.org/jira/browse/ASTERIXDB.

Table VI: The Algebricks Rules

RULE NAME	ACTION
BreakSelectIntoConjunctsRule	two selects conjuncted are broken to two selects
ByNameToByIndexFieldAccessRule	transforms names to positions
CheckFilterExpressionTypeRule	checks filter condition are of type boolean
ComplexJoinInferenceRule	removes subplan and generates join from two unnest calls
ComplexUnnestToProductRule	unnest followed by a join turns into two consecutive joins.
ConsolidateAssignsRule	assigns are gathered to one
ConsolidateSelectsRule	selects are gathered to one
ConstantFoldingRule	replaces one function by another
CountVarToCountOneRule	index replacement, from index to int
EliminateGroupByEmptyKeyRule	removes an empty group-by
EliminateSubplanRule	removes the wrapper- subplan call
EnforceOrderByAfterSubplan	order-by is pushed after the subplan
EnforceStructuralPropertiesRule	handles the ordering, grouping and partitioning properties and generates physical operators accordingly
ExtractCommonExpressionsRule	redundant expressions are removed
ExtractCommonOperatorsRule	expressions redundant are removed
ExtractDistinctByExpressionsRule	simplifies distinct by adding assigns
ExtractFunctionsFromJoinConditionRule	extracts the function-call to an assign
ExtractGbyExpressionsRule	extracts the expressions from group-by and pushes them to an assign
FuzzyEqRule	checks the condition in select or join operator. If it finds "∼=", it converts it into similarity-jaccand or edit-distance function.
IfElseToSwitchCaseFunctionRule	adds a switch case
InferTypesRule	computes outputs types of each operator (usually propagate)
InlineAssignIntoAggregateRule	gathers the assign into an aggregate call
InlineSingleReferenceVariablesRule	removes assign
InsertOuterJoinRule	eliminates subplan
InsertProjectBeforeUnionRule	if the variables are not used project them out
IntroduceAggregateCombinerRule	gathers aggregates into one
IntroduceDynamicTypeCastRule	casts using additional assign
IntroduceGroupByCombinerRule	adds a group-by
IntroduceGroupByForSubplanRule	subplan turns into group-by and smaller subplan
IntroduceMaterializationForInsertWithSelfScanRule	adds materialization
IntroduceProjectsRule	adds a projects
IntroduceRapidFrameFlushProjectAssignRule	optimization of frame consuming
IntroduceSelectAccessMethodRule	data-scan is replaced by a unnest-order-unnest-assign
IntroduceStaticTypeCastForInsertRule	casting of open types to closes, of bags to int etc
IntroduceUnnestForCollectionToSequenceRule	removes collection-to-sequence from plan
IntroHashPartitionMergeExchange	hash partition exchange operator followed by a sort merge is transformed into hash partition merge exchange
IntroJoinInsideSubplanRule	nested in subplan turns to a join true (the rewritten join plan will be done by a subsequent rule)
IsolateHyracksOperatorsRule	adds one to one exchange
LeftOuterJoinToInnerJoinRule	select + left-outer join are transformed to inner join
NestedSubplanToJoinRule	eliminates subplan and adds a join, in the case where the subplan does not get any variables from outside of the subplan
NestGroupByRule	removes subplan and lower groupby
PullSelectOutOfEqJoin	joins condition is split to a select and a join
PushAggFuncIntoStandaloneAggregateRule	push down aggregate function call
PushAggregateIntoGroupbyRule	pushes count into groupby and removes listify
PushAssignBelowUnionAllRule	pushes assign down
PushAssignDownThroughProductRule	simply pushed the assign earlier
PushFieldAccessRule	removes assign
PushFunctionsBelowJoin	pushes functions below a join
PushGroupByThroughProduct	operator replacement
PushLimitDownRule	generates a limit earlier in the plan
PushNestedOrderByUnderPreSortedGroupByRule	consolidates orderings
PushProjectDownRule	early projects
PushProperJoinThroughProduct	operator replacement
PushSelectDownRule	pushes select down
PushSelectIntoJoinRule	adds the select condition to join and removed the select operator.
PushSimilarityFunctionsBelowJoin	extends PushFunctionsBelowJoin
PushSubplanWithAggregateDownThroughProductRule	complex,pushes subplan down
ReinferAllTypesRule	recomputes types or similar
RemoveRedundantGroupByDecorVars	operator replacement
RemoveRedundantListifyRule	removes listify
RemoveRedundantVariablesRule	redundant variables are replaced by a unique reference
RemoveUnusedAssignAndAggregateRule	removes assigns
RemoveUnusedOneToOneEquiJoinRule	removes joins
ReplaceSinkOpWithCommitOpRule	replace sink with commit
SetAlgebricksPhysicalOperatorsRule	translates the logical to physical operators
SetAsterixPhysicalOperatorsRule	introduces BTree indexes
SetClosedRecordConstructorsRule	open-record-constructor becomes closed-record-construcotr if all the branches below lead to dataset scans for closed record types
SetExecutionModeRule	modifies the execution property of all operators
SimilarityCheckRule	looks for functions similarity-jaccard or edit-distance, and converts them to their equivalent check functions
SimpleUnnestToProductRule	two consecutive data-scans are transformed to a join (true)
SubplanOutOfGroupRule	not fired in Asterix test suite
UnnestToDataScanRule	transforms unnest to data-scan