UNIVERSITY OF CALIFORNIA,
IRVINE

Implementation and Analysis of the GroupJoin Operator in the ASTERIX BDMS

THESIS

submitted in partial satisfaction of the requirements
for the degree of

MASTER OF SCIENCE

in Computer Science

by

Manish Honnatti

Thesis Committee:
Professor Michael J. Carey, Chair
Professor Chen Li
Professor Harry Xu

2012

DEDICATION

To the patriarchs and the teachers.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to thank Professor Carey, first, for giving me the opportunity to work with the ASTERIX ecosystem. Second, and infinitely more important, I would like to thank him for his support, guidance and encouragement, which has some times gone beyond the realm of academia. I will always be grateful for the role he has played in my life at UC Irvine.

I must thank Professors Li and Mehrotra, for their guidance in the classroom and outside it. Much have I learned through and from them.

I would like to thank Professor Xu for agreeing to be on my committee at the last minute.

I would also like to thank the members of the ASTERIX team for letting me tinker and mess around with the code and for helping me in return. I would specifically like to mention Vinayak Borkar, Alexander Behm and Yingyi Bu for their unending help and support.

The initial exploration of the idea in this thesis was done in collaboration with my good friend and frequent project partner, Manik Sikka. I gratefully acknowledge his contribution in getting the proof of concept in decent shape.

To all my family, friends and well wishers, who are no doubt sighing in relief as this thesis work draws to its conclusion - my most sincere thanks for making this possible.

Semper gratiam habeo.

ABSTRACT OF THE THESIS

Implementation and Analysis of the GroupJoin Operator
in the ASTERIX BDMS

By

Manish Honnatti

Master of Science in Computer Science

University of California, Irvine, 2012

Professor Michael J. Carey, Chair

In database management systems with declarative query languages, Joins are arguably the most essential operators. The choice of Join algorithm often times influences costs (both I/O and time) by several orders of magnitude. In addition to Joins, Grouping operators are frequently used for generating summarized views and interpretations of data. Given commonly used data models and schemas, Grouping operators often have one or more Join operators preceding them in query plans for analytical queries. The GroupJoin operator has recently been proposed as a combination of the Join and Grouping operators and their associated algorithms. The goal of introducing GroupJoin is to eliminate a significant portion of the costs incurred when using Join and Grouping operators sequentially. This thesis investigates implementation details and usage scenarios with respect to GroupJoin in the context of the ASTERIX BDMS. Further, a comparative performance evaluation is carried out and the results are examined. In conditions of equivalence with the standard Join and Group queries, the experiments reported here for the GroupJoin operator show a factor of improvement in performance of up to 1.23.

Chapter 1

Introduction

Most data models and schemas used in the real world have data normalized, and thus spread over multiple tables, leading to the recurrent use of Joins in meaningful queries. The Grouping operator is also frequently used, especially in systems such as analytics engines. Though some specialized systems do maintain aggregate views of data at frequent intervals, in general, Grouping queries often contain one or more Join operator(s).

In queries involving both Joins and Grouping, a significant improvement in costs can be achieved by the use of a merged operator in which both the Join and Grouping (and related aggregation) are done together [1, 2]. This operator, called a GroupJoin, reduces costs due to two important improvements:

1. The full result of the Join operator need not be generated and, crucially, need not be written/read from the disk.
2. A typical hash-based Join followed by Grouping involves the creation of two hash tables – one for each operator. The GroupJoin, with its combined operation,

requires only one hash table, reducing the query's memory requirement.

This thesis reports on an effort to implement GroupJoin as a new operator in the ASTERIX BDMS [3]. In this effort, hash-based operators were investigated, i.e. Hash GroupJoin operators were created and compared with the use of separate Hash Join and Hash Grouping operators. As discussed later on, the GroupJoin operator's advantage is significant in the case of large ad-hoc equi-joins, for which the Hash Join is generally favored.

A brief note on the history of the GroupJoin is in order. von Bultzingsloewen [1] seems to be the inventor of the operator with the creation of the outer aggregation. Several others have called the operator by different names until the latest, i.e. GroupJoin, gained acceptance [2]. The operator has thus existed for at least two decades, though traditional RDMSs don't seem to have incorporated it as of yet.

The rest of this thesis is structured as follows. In Section 2, a general introduction to the GroupJoin operator and the notation used are made. Equivalence conditions between GroupJoin and Join+Group are introduced. A detailed hash-based algorithm for the GroupJoin is also described. In Section 3, key implementation details with respect to ASTERIX are discussed. Section 4 is devoted to an experimental performance analysis of GroupJoin – this includes descriptions of the infrastructure setup and the analysis methodology, and also the results along with their interpretation.

Chapter 2

The GroupJoin Operator

In this Chapter we look at the GroupJoin operator in detail. Section 2.1 consists of a description of the notation used in this thesis and also a broad look at the use and value of the GroupJoin operator. In Section 2.2 we lay out the conditions under which the GroupJoin operator is considered equivalent to a sequence of a Join and a Grouping. These conditions must be met before the GroupJoin can be introduced into any query plan. A general algorithm for the GroupJoin operator is discussed in Section 2.3.

2.1 Basics

We begin by introducing the notation to be used in the rest of this thesis. The notation used here is essentially based on the standard relational algebra notation used in general DB theory [4, 5]. Table 2.1 lists the notation used.

Symbol	Description
$\boldsymbol{\sigma}_p$	Selection with predicate p
$\boldsymbol{\pi}_{a1,a2,\ldots,ak}$	Projection of attributes *a1, a2, ..., ak*
$\bowtie_j$	Inner Join over join predicate j
$⟕_j$	Left Outer Join over join predicate j
$\boldsymbol{\gamma}_{g1,g2,\ldots,gm;A1,A2,\ldots An}$	Grouping over attributes *g1, g2, ..., gm*
	and aggregation with functions *A1, A2, ..., An*
$\boldsymbol{\tau}_l$	Sort by attribute l

Table 2.1: Notation used in this thesis

Further, the following symbol for GroupJoin is also introduced:

$$\bowtie\!\ulcorner_{g1,g2,\ldots,gm;A1,A2,\ldots An}$$

Here, *g1, g2, ..., gm* are the Grouping attributes

and, *A1, A2, ..., An* are the Aggregation functions applied on a set of attributes.

To show the use and value of the GroupJoin operator, consider Query 13 from the TPC-H [6] benchmark query set shown in Figure 2.1 (Query 13 in ASTERIX Query Language, i.e., AQL, can be found in Section 4.2). This query computes the distribution of customers by the number of orders they have placed, i.e., a count-of-counts query.

```
select
    c_count, count(*) as custdist
from
    (
        select
            c_custkey, count(o_orderkey)
        from
            customer left outer join orders on
                c_custkey = o_custkey
                and o_comment not like '%special%requests%'
        group by
            c_custkey
    ) as c_orders (c_custkey, c_count)
group by
    c_count
order by
    custdist desc, c_count desc;
```

Figure 2.1: TPC-H Query 13

In the absence of the GroupJoin, an optimized query plan generated by a typical RDBMS engine is shown in Figure 2.2.

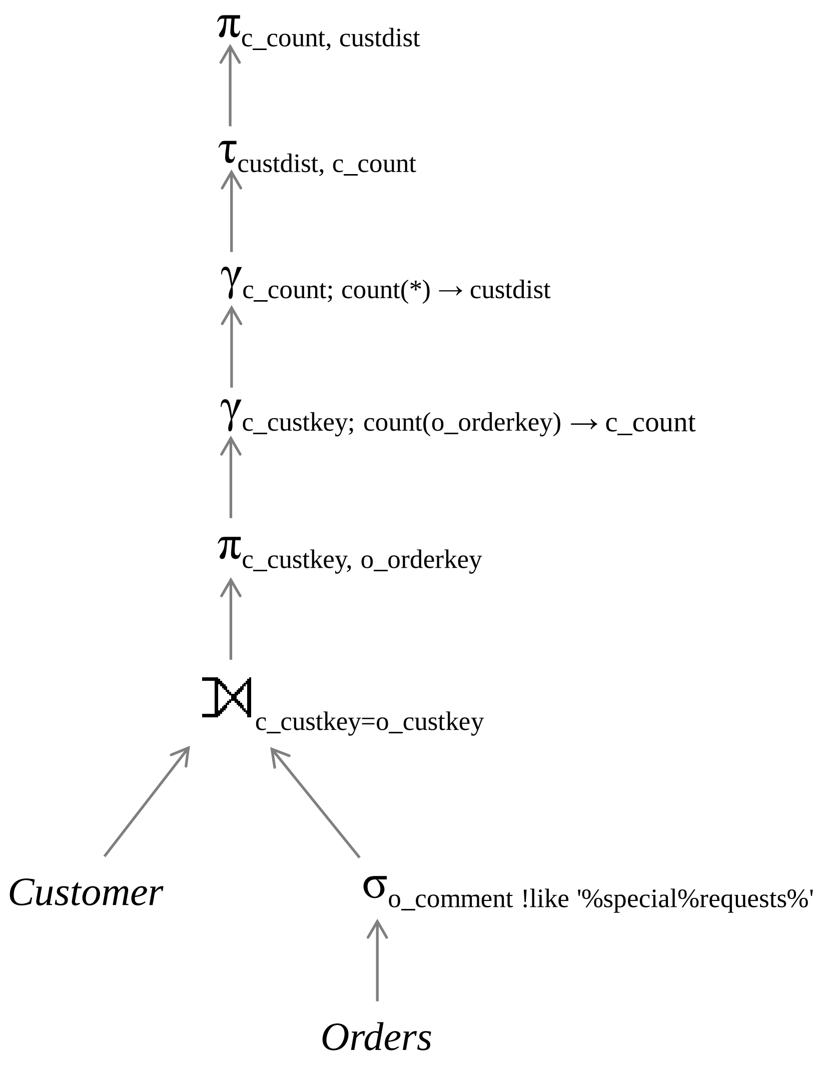
π c_count, custdist
τ custdist, c_count
γ c_count; count(*) → custdist
γ c_custkey; count(o_orderkey) → c_count
π c_custkey, o_orderkey
⟕ c_custkey=o_custkey
Customer
σ o_comment !like '%special%requests%'
Orders

Figure 2.2: Join+Group Plan for TPC-H Q13

In this plan, the result of the Left Outer Join, ⟕, is fed to the first Grouping operator, γ, after Projecting the required fields. With large data sets, it is highly likely that some of the join results will be written to disk. When the GroupJoin is introduced in this query, it would take the place of the Left Outer Join and the Grouping operator. In this example, i.e., Query 13, the GroupJoin can be written in terms of the Left

Outer Join and the subsequent Grouping as:

$$customer \bowtie\!\!\ulcorner_{c\_custkey;count(o\_orderkey) \rightarrow c\_count}\ orders =$$
$$\gamma_{c\_custkey;count(o\_orderkey) \rightarrow c\_count}(\pi_{c\_custkey,o\_orderkey}(customer ⟕_{c\_custkey=o\_custkey}\ orders))$$

This will yield the query plan as shown in Figure 2.3.

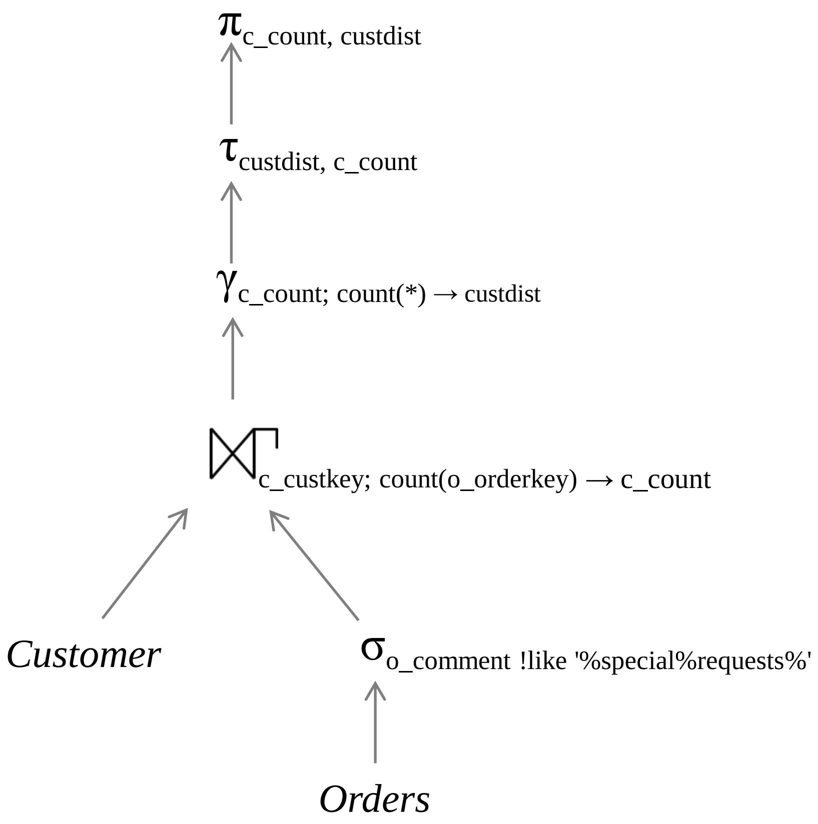
π c_count, custdist
τ custdist, c_count
γ c_count; count(*) → custdist
c_custkey; count(o_orderkey) → c_count
Customer
σ o_comment !like '%special%requests%'
Orders

Figure 2.3: GroupJoin Plan for TPC-H Q13

2.2 Equivalence Conditions

With the GroupJoin operator having to replace a sequence of Join and Group operators, it is necessary to identify the equivalence conditions for such a replacement. In particular, it is to be noted that not every sequence of Join and Group operators can be replaced by the Groupjoin operator.

The conditions that must be satisfied to ensure the GroupJoin operator produces the same results as the operators it replaces are discussed in the rest of this section. For incorporation into the ASTERIX query optimizer, these conditions have been adapted from the equivalence rules presented in [2]. Generally speaking, the equivalence conditions ensure that the GroupJoin is able to maintain the same Grouping granularity as that of the operators it is going to replace. We will also look at the case when the GroupJoin's granularity is smaller than the Join+Grouping in the query plan, and how the GroupJoin can still be viable for the plan in such a case.

2.2.1 Condition 1: Maintaining Grouping Granularity w.r.t. Join and Group Attributes

The GroupJoin operator, being an amalgam of the Join and Grouping operators, both joins and groups its input tuples at the same time. For this to happen, the simplest criterion is to have both **the join and the grouping predicates be over the same set of attributes**. To illustrate, consider the relations L and R, and a simple query shown in Figure 2.4. The result of Sample Query 1 is also shown in Figure 2.4.

The two alternative plans with Join+Group and GroupJoin are shown in Figure 2.5. For this query, having an Inner Join operator with predicate L.L2=R.R1 and

L		R	
L1	L2	R1	R2
1	1	1	1
1	2	1	3
2	3	2	5
2	4	2	7

```
select L.L2, sum(R.R2) as sumcol
from L,R
where L.L2=R.R1
group by L.L2
```

Result	
L2	sumcol
1	4
2	12

Figure 2.4: Illustrative relations L & R; Sample Query 1; Result

a Grouping operator over L.L2, the result has just 2 groups since two of the fields in L.L2 (3, 4) have no join match from R.R1. The granularity is maintained by GroupJoin insertion because the grouping and join attributes are the same.

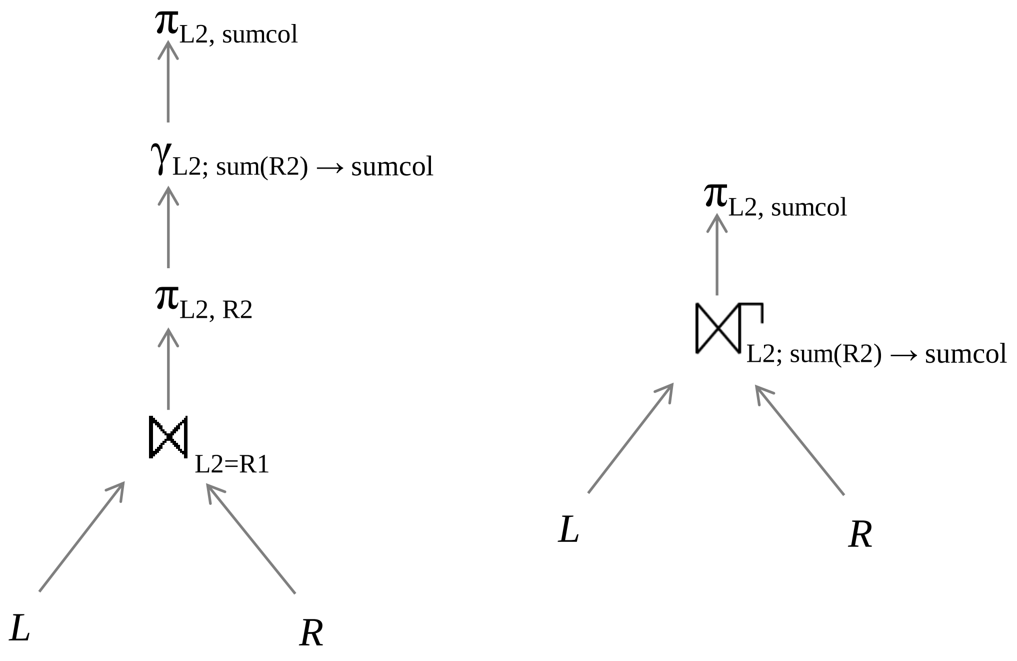

Figure 2.5: Query plans with Join+Group and GroupJoin for Sample Query 1

NOTE: The effects of a Primary Key in the Join/Grouping attributes are discussed in Section 2.2.4 as it also concerns the equivalence condition discussed in the next sub-section.

In case the above attribute match condition is not met, and the granularity of grouping over the Join attributes is finer than the granularity of grouping over the Group attributes, the GroupJoin operator can still be introduced. However, an additional Grouping operator must also then be introduced (or modified, if one already exists) after the GroupJoin, such that it produces groups of the same granularity as in the original plan. Also, the Aggregate function(s) must be decomposable for this to be possible [7]. For example, consider a Grouping operator, γ, with a *count* aggregate function being split into two Grouping operators, γ_1 and γ_2, such that γ_1 feeds γ_2 in the plan. The aggregate function in γ_1 continues to be *count.* However, the aggregate function in γ_2 needs to be *sum* to ensure that the same results are obtained. Decompositions of various aggregate functions are presented in Appendix A.

To illustrate this type of equivalence concretely, consider again the relations L and R and a new query (Sample Query 2) shown in Figure 2.6.

L		R	
L1	L2	R1	R2
1	1	1	1
1	2	1	3
2	3	2	5
2	4	2	7

```
select L.L1, sum(R.R2) as sumcol
from L,R
where L.L2=R.R1
group by L.L1
```

Result	
L1	sumcol
1	16

Figure 2.6: Illustrative relations L & R; Sample Query 2; Result

For Sample Query 2, the Join predicate is L.L2=R.R1 but Grouping is over L.L1. Replacing the Join and Grouping operators with only a GroupJoin in this query would produce groups based only on L.L2. Thus, an additional Grouping operator also needs to be introduced after the GroupJoin. The two alternative plans for Sample Query 2 are shown in Figure 2.7. Attention is drawn to the Grouping operator $\gamma_{L1;sum(tmpcol)\rightarrow sumcol}$ in the GroupJoin plan in the right half of Figure 2.7. This Grouping operator ensures that the level of grouping, even with the GroupJoin operator in the plan, is maintained at L.L1.

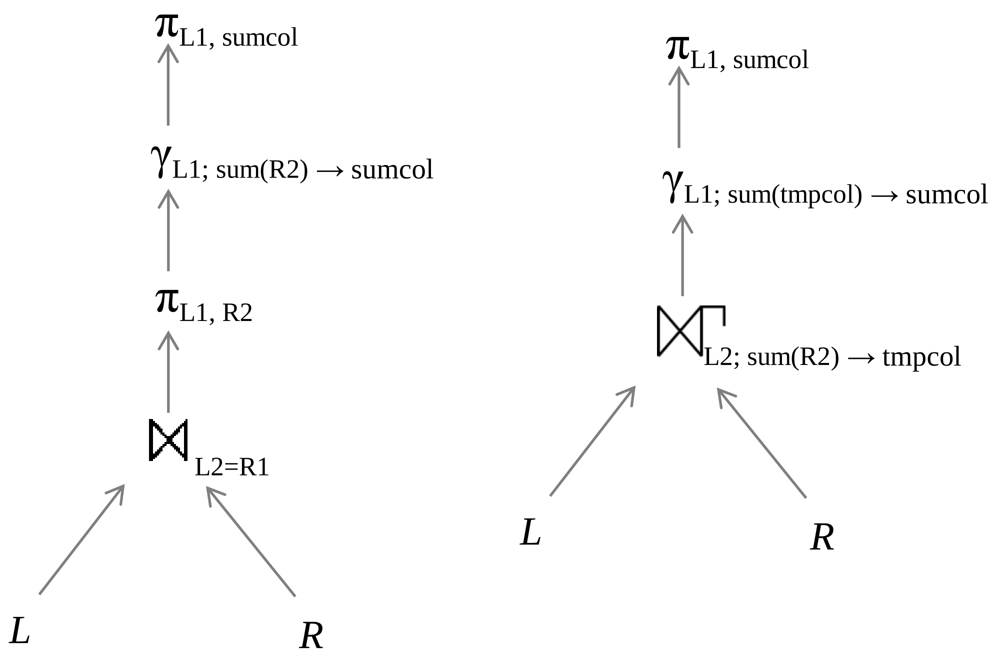

Figure 2.7: Query plans with Join+Group and GroupJoin for Sample Query 2

2.2.2 Condition 2: Maintaining Grouping Granularity w.r.t. Join Cardinality

The Cardinality of the Join needs to be one-to-many or one-to-one, i.e., each tuple from the right relation, R, must join with only one tuple in the left relation, L. This is necessary because the GroupJoin, like a traditional hash join, creates one group for each tuple in L. Thus, for a many-to-one or many-to-many Join a tuple from R joins with more than one tuple in L, i.e., it would go into more than one group. Using the GroupJoin in such a query would not produce the same number of groups as when the Join and Group are used.

To illustrate this, consider relations L and R, and Sample Query 3 shown in Figure 2.8.

L		R	
L1	L2	R1	R2
1	1	1	1
1	2	1	3
2	3	2	5
2	4	2	7

```
select L.L1, sum(R.R2) as sumcol
from L,R
where L.L1=R.R1
group by L.L1
```

Join+Group Result	
L1	sumcol
1	8
2	24

GroupJoin Result	
L1	sumcol
1	4
1	4
2	12
2	12

Figure 2.8: Illustrative relations L & R; Sample Query 3; Result

It can be seen that the GroupJoin produces more groups and, thus, cannot be introduced into the plan, as this rewrite would yield incorrect results.

The presence of the *unique constraint* on any of the attributes from the left relation used in the Join **and** the Grouping predicates would be an example of satisying this condition. The case of the unique constraint being enforced as part of a Primary Key is discussed in Section 2.2.4.

2.2.3 Condition 3: Aggregation after Group creation

The aggregate attributes, i.e. attributes to which Aggregation functions in γ are applied, must be in the right relation, R. This condition is necessary as the functions are applied after the groups are created and as such, aggregating over attributes from the left relation, L is not possible.

2.2.4 Notes on Equivalence

Having laid out the equivalence conditions, the following are some general notes on introducing GroupJoin into a query plan:

1. *Effect of Primary Key in Join/Group attributes*: The presence of a Primary Key(PK) from the left input relation in the Join/Group attributes implies that Condition 2 described in Section 2.2.2 will be met. The PK enforces the *unique constraint* which implies either one-to-many or one-to-one join cardinality.

   Further, having a PK also ensures that, regardless of the other grouping attributes, the grouping granularity will be maintained with GroupJoin introduction as long as the Join attributes are a subset of the Grouping attributes (including the PK). This is because a PK is the finest possible granularity that can achieved in any relation, i.e., at each row. To illustrate this, consider these Grouping attributes: $\{g_1, g_2, g_3, PK, g_4, g_5\}$. Attributes g_1, g_2, g_3, g_4 and g_5 have absolutely no effect on the Grouping granularity as their granularity cannot be finer than that of the PK. Thus, if the Join attributes are $\{g_1, g_2, g_3, PK\}$ the GroupJoin can be introduced in the query.

2. *Outer Join Nature*: If the Join in question is of the Left Outer type, the GroupJoin can be introduced, itself also being of the Left Outer type. Right Outer Joins disqualify the GroupJoin as the formation of NULL groups would then be possible. Figure 2.9 shows an example with a Left Outer Join.

L			R	
L1	L2		R1	R2
1	1		1	1
1	2		1	3
2	3		2	5
2	4		2	7

```
select L.L2, sum(R.R2) as sumcol
from L Left Outer Join R
     on L.L2=R.R1
group by L.L2
```

Result	
L2	sumcol
1	4
2	12
3	null
4	null

Figure 2.9: Illustrative relations L & R; Sample Query 4; Result

2.3 GroupJoin Algorithm

The Hash-based GroupJoin algorithm studied in this thesis consists of three major phases. We begin by describing its behavior when the grouped result fits in memory. For Sample Query 4 shown in Figure 2.9, the algorithm would work as follows.

Phase 1, Build: In this first phase of GroupJoin, all tuples from L are read page by page from disk, hashed on L.L2 and inserted into an in-memory hash-table. This is shown in Figure 2.10. (At this point each tuple's Aggregate state is uninitialized.)

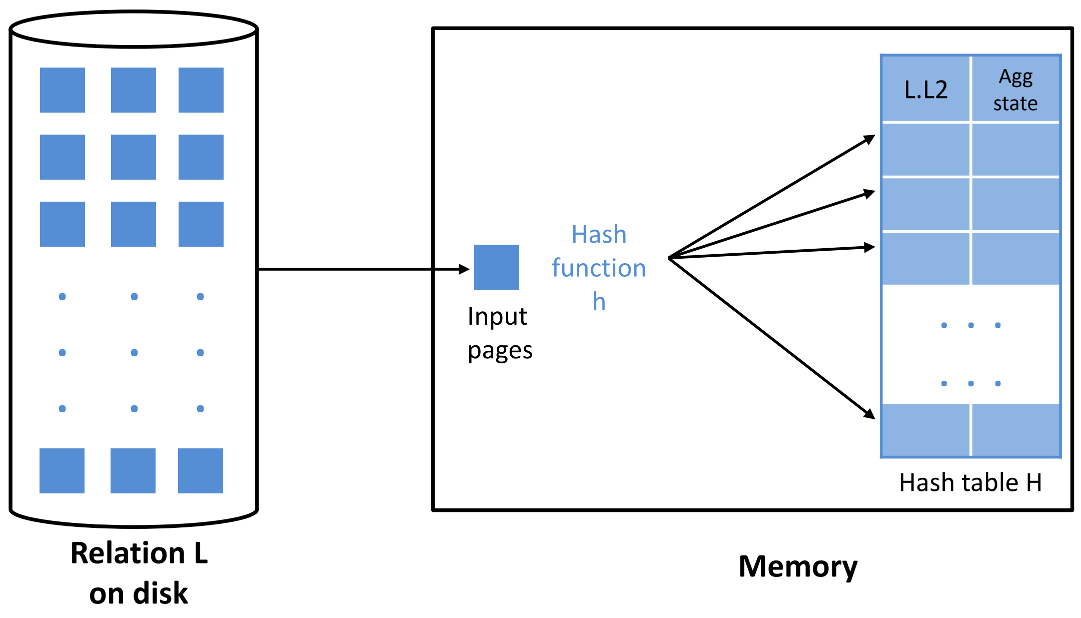

Figure 2.10: Build phase

Phase 2, Probe: In the second phase, each tuple from R is hashed on R.R1 and probes the hash-table generated in the first phase. If a match is found, i.e. join predicate L.L2=R.R1 is satisfied, then the corresponding aggregate state, *s1* is initialized and updated based on R.R2. This is shown in Figure 2.11.

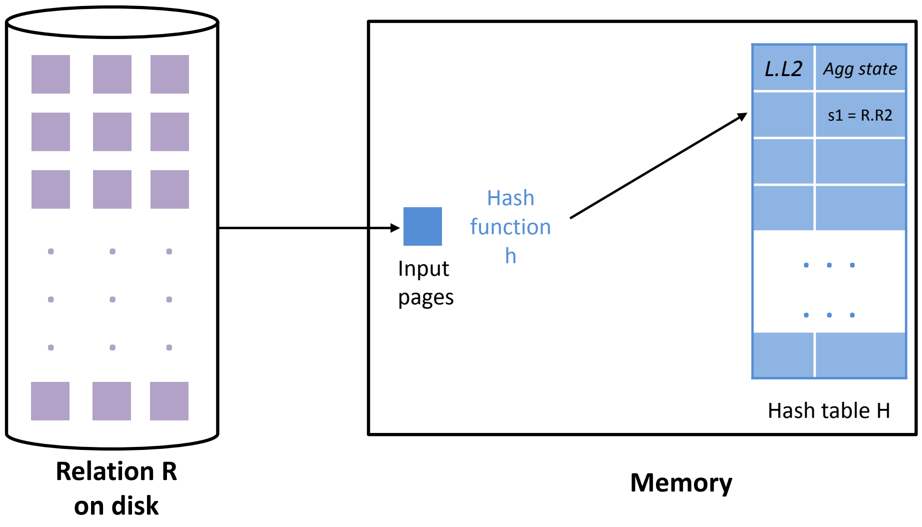

Figure 2.11: Probe phase with Aggregate state initialization

When a tuple from R joins with a tuple from L that has previously had an aggregate state initialized, the aggregate state is simply updated with R.R2. This is shown in Figure 2.12. If no match is found, the tuple from R is discarded.

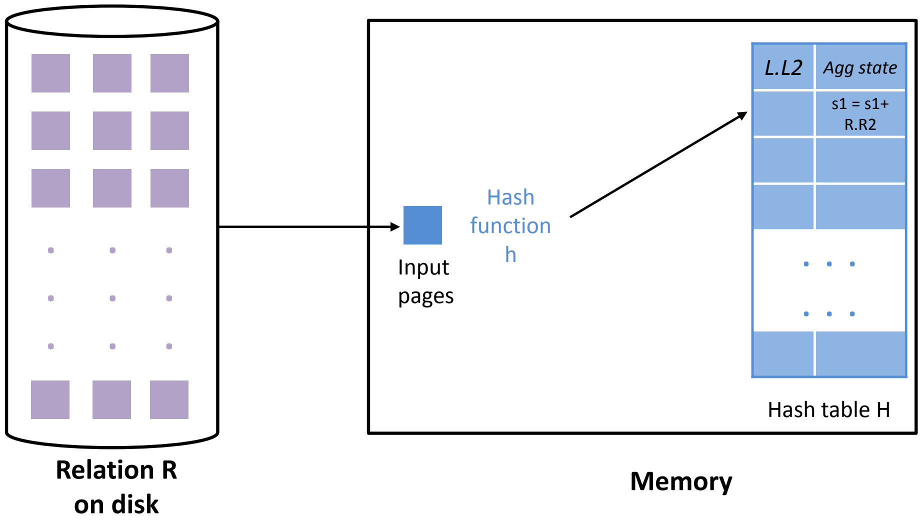

Figure 2.12: Probe phase with only Aggregate state update

Phase 3, Output: In the third and last phase, final aggregate processing, if needed, is performed (e.g. for average, where the sum is divided by the count). Null-processing as defined in the engine is also done. For an Outer Join, since there can be tuples in L that do not have any match from R, they need to be present in the GroupJoin output with an aggregate value of NULL. Instead, if the query contained an Inner Join, the unmatched tuples in L will be discarded. Figure 2.13 shows the output phase with the example of Sample Query 4.

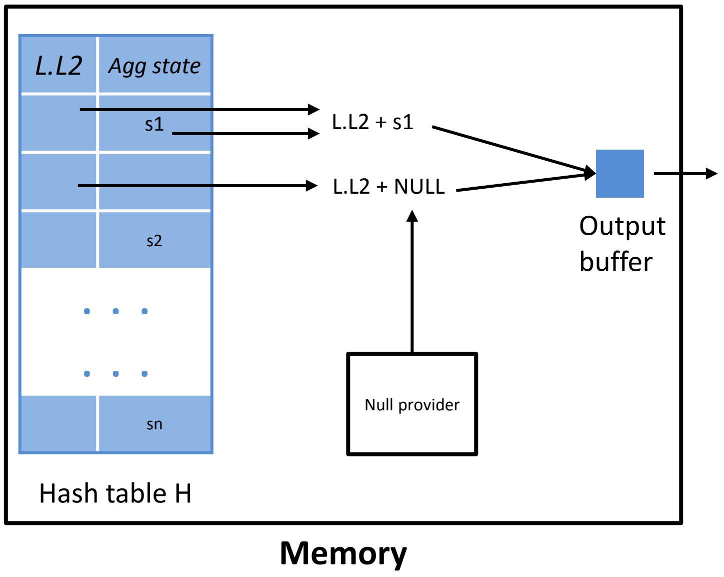

Figure 2.13: Output phase

The general GroupJoin algorithm can be described as shown in Figure 2.14. L and R are the left and right input relations respectively. j_L is the set of attributes from L found in join predicates while j_R is the set of join attributes from R. g is the set of Grouping attributes and a is the set of Aggregate attributes (from R). *s(t)* represents the Aggregate state for a tuple t.

```
for each tuple t in L:
    insert into hash-table, H:
        h(t.jL),
        t.g
end for
for each tuple t in R:
    probe h(t.jR) in H
        if h(t.jR) exists in H:
            if not s(t).isInitialized:
                s(t).initialize
            end if
            s(t).update(t.a)
        end if
end for
for each tuple t in H:
    if s(t) != NULL  or (s(t) == NULL and Left Outer Join):
        s(t).finalize
        output:
            t,
            s(t)
    end if
end for
```

Figure 2.14: Algorithm for GroupJoin operator

In describing the GroupJoin algorithm so far we have focused on the case when the grouped result fits in memory. For input data sets which cannot be fit in memory a Hybrid Hash GroupJoin is well suited. Hybrid Hash GroupJoin works similar to a Hybrid Hash Join, i.e., both the input datasets are partitioned into B+1 blocks and GroupJoin is performed on each pair of blocks. While partitioning the left input relation the first block is retained in memory in the form a hash table. While partitioning the right input relation, tuples that would go into its first block are directly probed in the hash table and a probe phase like in the in-memory operator is executed. When all the tuples from the first block have been processed, the output phase from the in-memory operator is executed. This technique is applied to all the block pairs in sequence. A detailed discussion of the Hybrid Hash GroupJoin operator, as implemented in ASTERIX, can be found in Section 3.2.1.

Chapter 3

ASTERIX Implementation

This Chapter deals with the implementation of GroupJoin in ASTERIX, starting with an overview of the ASTERIX stack. We will then discuss the implementation details for GroupJoin with respect to each layer of the stack.

3.1 About ASTERIX

The ASTERIX project at UC Irvine began in early 2009 with the objective of creating a new parallel, semistructured information management system. The ASTERIX software stack consists of three distinct and reusable architectural layers - ASTERIX, Algebricks and Hyracks. These layers are summarized in Figure 3.1. This section gives a brief overview of each layer in the ASTERIX stack [8].

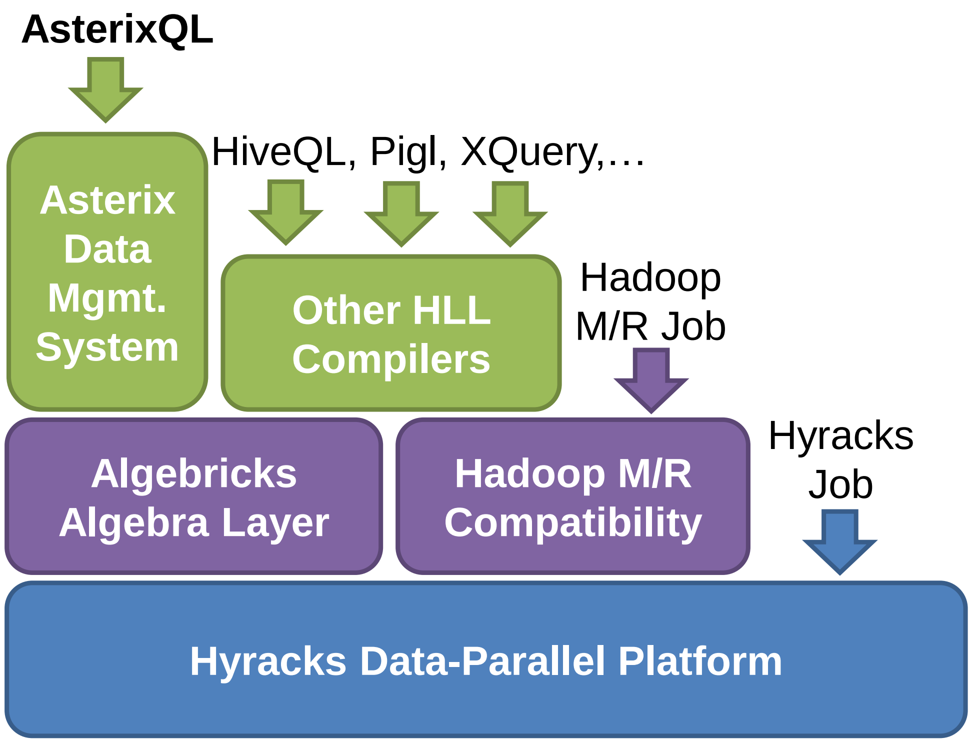

Figure 3.1: The ASTERIX Software Stack

3.1.1 Hyracks Runtime Layer

The Hyracks layer of ASTERIX [9] is the bottom-most layer of the stack. Hyracks is the runtime layer whose job is to accept and manage data-parallel computations requested either by direct end-users of Hyracks or by the layers above it in the ASTERIX software stack.

Jobs are submitted to Hyracks in the form of directed acyclic graphs that are made up of *Operators* and *Connectors*. *Operators* are responsible for consuming partitions of their inputs and producing output partitions. *Connectors* perform redistribution of data between different partitions of the same logical dataset. Hyracks includes a library of Operators and Connectors, with various Hash-Join and Grouping operators being among them.

3.1.2 Algebricks

Algebricks is a model-agnostic, algebraic layer for parallel query processing and optimization [10]. Having its origin as the center of the AQL compiler and optimizer of the ASTERIX system, Algebricks was eventually reborn as a public layer in its own right. Rewrite rules that apply commonly across data models, including standard rewrites from relational algebra and various rewrites for partitioning and parallelism, live at this level of the stack.

To be useful for implementing arbitrary languages, Algebricks has been carefully designed to be agnostic of the data model of the data that it processes. Logically, operators operate on collections of tuples containing data values. The data values carried inside a tuple are not specified by the Algebricks toolkit; the language implementor is free to define any value types as abstract data types.

3.1.3 ASTERIX

The topmost layer of the ASTERIX software stack is the ASTERIX parallel information management system [11]. Data in ASTERIX is based on a semistructured data model. As a result, ASTERIX is well-suited to handling use cases ranging from rigid, relation-like data collections, whose types are well understood and invariant, to flexible and potentially more complex data where little is known ahead of time and the instances in data collections are highly variant and self-describing. The ASTERIX data model (ADM) is based on borrowing the data concepts from JSON [12] and adding additional primitive types as well as type constructors borrowed from object databases [13, 14].

ASTERIX queries are written in AQL (the ASTERIX Query Language), a declarative query language designed by taking the essence of XQuery [15], most importantly its FLWOR expression constructs and its composability, and then simplifying and adapting it to query the types and modeling constructs of ADM.

ASTERIX compiles an AQL query into an Algebricks program. This program is then optimized via algebraic rewrite rules that reorder the Algebricks operators as well as introducing partitioned parallelism for scalable execution, after which code generation translates the resulting physical query plan into a corresponding Hyracks job. The resulting Hyracks job uses the operators and connectors of Hyracks to compute the desired query result.

3.2 GroupJoin in the ASTERIX Stack

The GroupJoin operator in ASTERIX is an internal operator that the optimizer can introduce (where applicable) to improve performance. The implementation of Hash based GroupJoin operators was based on existing Hash Join and Hash Group operators. Implementation specifics as they apply to each layer of the ASTERIX stack are discussed in the rest of this section. An overview is shown in Figure 3.2

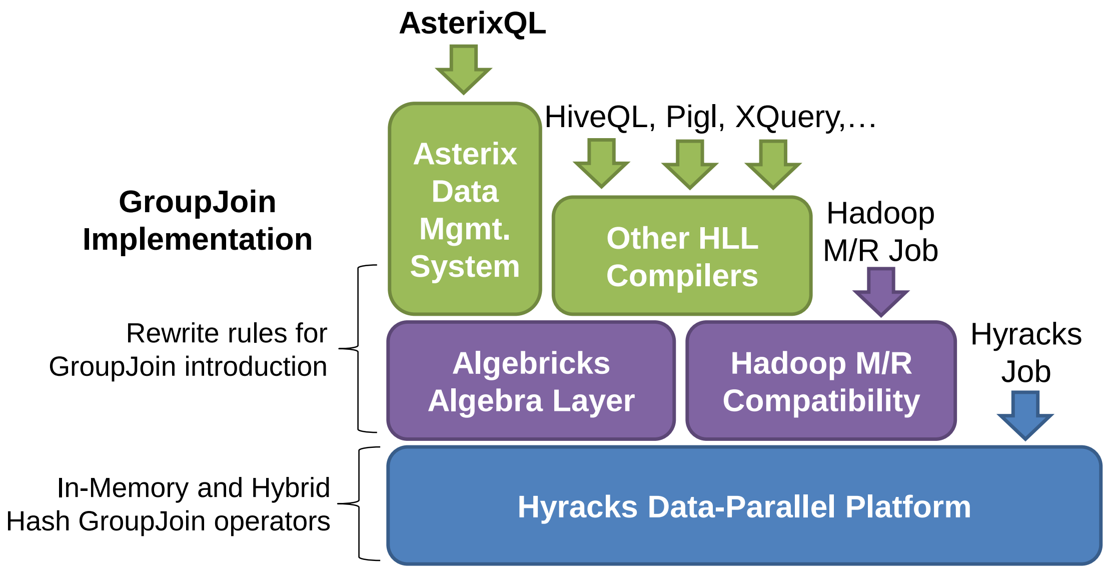

Figure 3.2: The ASTERIX Software Stack with GroupJoin Implementation specifics

3.2.1 GroupJoin in Hyracks

In the Hyracks runtime layer, two GroupJoin operators were implemented - an exclusive In-Memory Hash GroupJoin operator and a Hyrbrid Hash GroupJoin operator. The In-Memory operator is an implementation of the GroupJoin algorithm described in Section 2.3.

The Hybrid Hash GroupJoin operator runs very similarly to the Hybrid Hash Join algorithm found in [16]. First, in the *build phase*, the left input relation, say L, is partitioned into B+1 partitions by the use of a hash function. Tuples that go into the first partition L_0, are inserted into an in-memory hash table. Other tuples are written to their corresponding output buffer. Each buffer has a file on disk called a *bucket file*. Whenever an output buffer becomes full, it is flushed to its corresponding bucket file. A feature specific to the GroupJoin implementation is that along with each tuple's entry in the in-memory hash table an extra state pointer is also inserted.

This pointer is intended to point to the location of the aggregate state of that particular entry in a state buffer. Initially it is set to -1, as the aggregate state needs to be initialized only on the first join match in the probe phase. The build phase is shown in Figure 3.3.

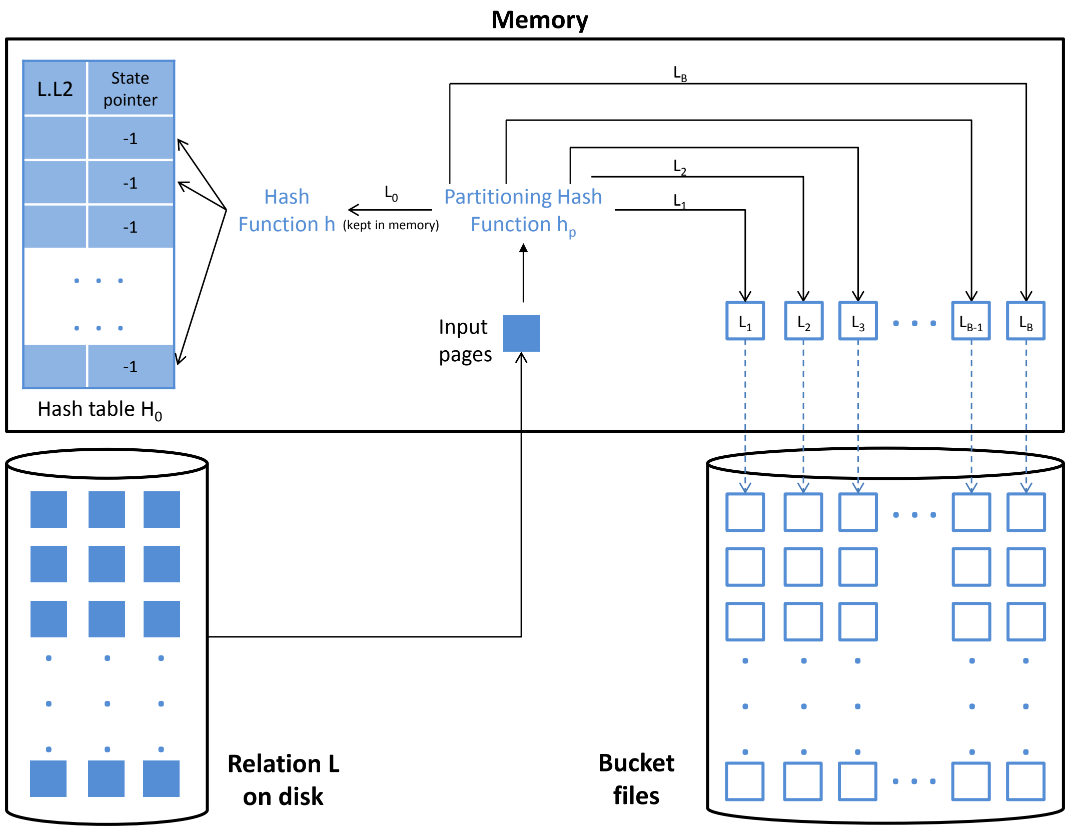

Figure 3.3: Hybrid Hash GroupJoin Build Phase

In the next phase, i.e. the *probe phase*, tuples from the right input relation, say R, are hashed with the same function used in the build phase. Tuples that go into partitions other than the first are written to the corresponding output buffers, much like in the build phase. For each tuple that goes into the first partition R_0, the hash table held in memory is probed. If a match is found, the corresponding aggregate state is retrieved and updated using the relevant attributes from the tuple being matched. If there is no aggregate state initialized, i.e. the state pointer is -1, an aggregate state is allocated in the state buffer and initialized with the attribute(s) from the tuple being matched. The pointer is also updated to point to the newly allocated state in the state buffer in this case. The probe phase is illustrated in Figure 3.4.

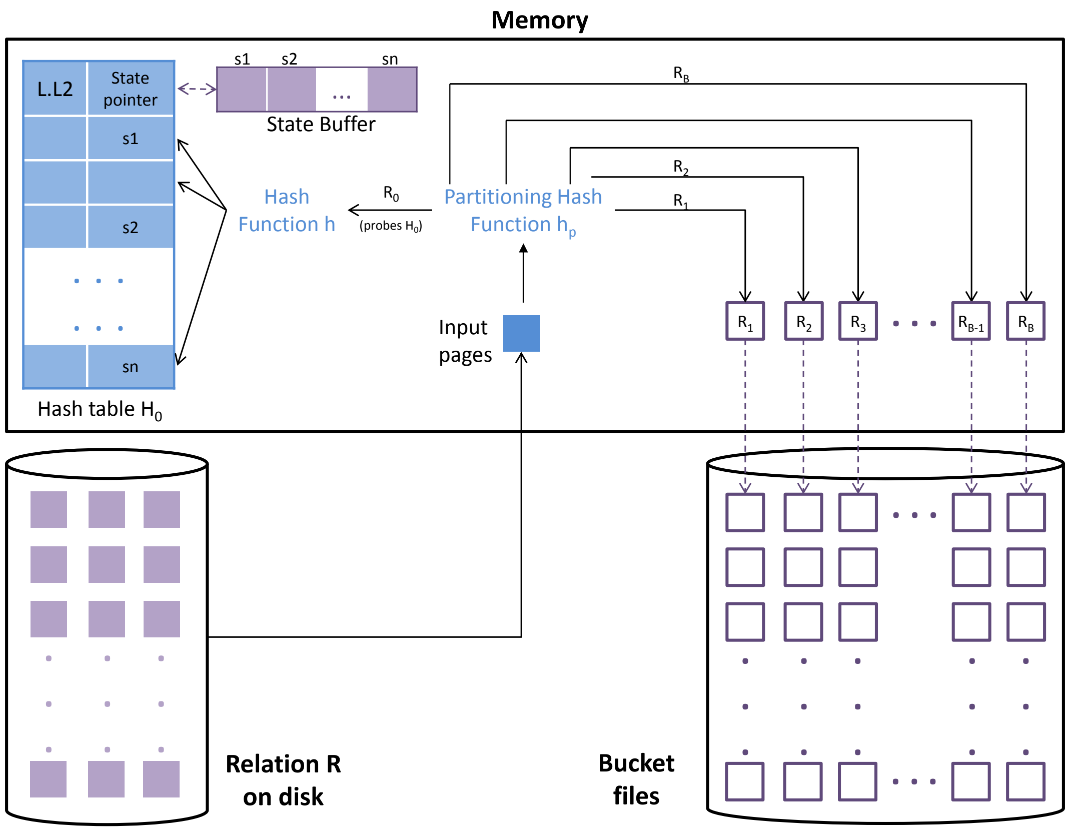

Figure 3.4: Hybrid Hash GroupJoin Probe Phase

Another difference with the Hybrid Hash Join is that tuple pairs are not immediately output when a hash table match is found in the probe phase. Since we are Grouping (and aggregating) while we are Joining, the Hybrid Hash GroupJoin outputs results only after all of the tuples belonging to R_0 have been processed. This necessitates a final output phase for the GroupJoin operator.

In the *output phase*, all the entries in the hash table are enumerated. For each entry, the tuple from partition L_0 is appended with the aggregate value from its corresponding aggregate state and then emitted. The outcome handling for *dangling tuples*, i.e., tuples that have had no join match, is dependent on the left outer nature specified for thr GroupJoin. For a left outer GroupJoin, the dangling tuple from L_0

is appended with null(s) as defined by the user or the AQL query from upper layers. For an inner GroupJoin, such tuples are simply discarded. The output phase for a left outer GroupJoin is shown in Figure 3.5.

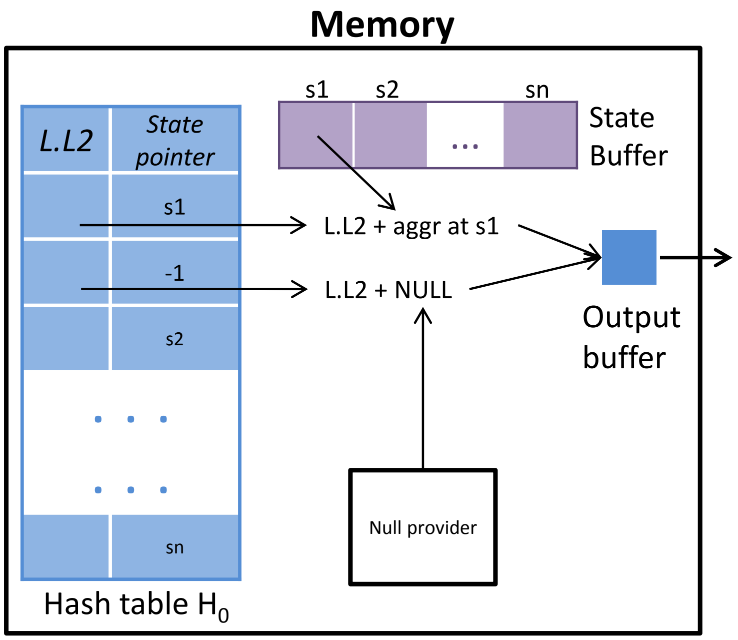

Figure 3.5: Hybrid Hash GroupJoin Output Phase

The build, probe and output phases described above are then applied to each of the remaining B partition pairs in sequence.

3.2.2 GroupJoin in Algebricks

To have Algebricks be able to insert GroupJoin in query plans where applicable, two different sets of implementation tasks were carried out. The first task was the creation of a logical GroupJoin operator and two physical GroupJoin operators - one each for the In-Memory and Hybrid Hash operator flavors. The second task was the implementation of the algebraic rewrite rules to determine the suitability of the GroupJoin operator for the query being optimized. These rewrite rules were an Algebricks-specific implementation of the equivalence conditions described in Section 2.2. It is worth mentioning that, though the optimizer in the ASTERIX project is rule-based, the GroupJoin is generally "safe" to insert, as its performance is at worst the same as that of the Join and Grouping operators that it replaces. (This worst case scenario can occur when the join cardinality is one-to-one and the Grouping attributes are the same as the Join attributes).

As part of the GroupJoin rewrite rule, the following checks are made sequentially, and at any check failure, the GroupJoin operator is deemed unsuitable for introduction into the query plan:

1. The most basic check is to match the pattern of a Join operator followed by Group operator.
2. Check if the Join can be a Hash-based Join, i.e. if it is an equi-join with scalar attributes in the join predicate. This check is necessary since we are dealing with only the Hash GroupJoin operator.
3. Check if the equivalence conditions as outlined in Sections 2.2.1 and 2.2.2 are satisfied, i.e. if the GroupJoin can be introduced without changing the granularity of the grouping. The simplest way to satisfy these conditions is the presence of a Primary Key in the Group and Join attributes. It should be mentioned that Functional Dependencies of attributes are also accounted for in these checks. For example, while Joining on attribute j, and Grouping on attribute g, the condition in 2.2.1 is deemed to have been met if j has a Functional Dependency on g.

4. Check if the condition outlined in Section 2.2.3 is satisfied, i.e. if the attributes used in the Aggregate functions come only from the right input relation.

5. A further "safety" check on the Aggregate functions is that they should be *compressive*. The presence of a non-compressive Aggregate function, like listify, could affect the performance of the GroupJoin adversely. This is because the aggregate state for these types of aggregations tends to occupy increasing space, and a growing hash-table in the GroupJoin would likely become prone to disk flushes and quickly negate any advantages of using the GroupJoin.

3.2.3 GroupJoin in ASTERIX

Since the GroupJoin is an internal operator, there were no significant implemenation necessities in the topmost ASTERIX layer. A minor feature that was implemented was the provision of a *query hint* (also known as *optimization hint* in some RDBMS products) in AQL, allowing the user to disable the GroupJoin. This is useful when only a regular Join operator is desired to be allowed in the query being hinted about.

3.2.4 Parallel Execution

Parallelization of the GroupJoin is quite similiar to the parallelization of hash-based Join operators. ASTERIX being a shared-nothing BDMS, parallelization involves horizontally partitioning the input datasets on the Grouping attributes using a hash function and redistributing the partitions over the nodes available in the network. (Recall that the Join attributes are a subset of the Grouping attributes, so no additional partitioning is required.) Then, each node locally runs either the In-Memory GroupJoin operator or the Hybrid Hash GroupJoin operator over its input partitions.

As such, the partitioning mechanism is not part of the GroupJoin operator.

The results are then merged and/or redistributed over the network as required in the query plan.

Chapter 4

Performance Evaluation

The experimental setup and the results of some initial performance comparisons of query execution times with and without GroupJoin are reported in this chapter. First, the hardware setup and the datasets used for the experiments are described. Next, the results of the experiments are laid out and performance improvements due to the GroupJoin are calculated. Finally, the results are investigated.

4.1 Infrastructure

For the experiments, a cluster of 4 IBM machines each with a 4-core Xeon 2.27 GHz CPU, 12GB of RAM, and 4 locally attached 10K rpm SATA drives was utilized. An additional master node used was a separate machine which acted as a coordinator only and did not participate in query execution. At each node, 10GB of the available 12GB memory was allocated for the ASTERIX software stack. The data sets were pre-loaded into B-tree storage. Each of the 4 available disks was utilized as a data store.

4.2 TPC-H Datasets & Queries

For the experiments we have chosen to focus on TPC-H's Query 13, since its Grouping attribute is the same as its Join attribute and it is thus a GroupJoin candidate. The goal of the experiments was to compare the query execution times with and without GroupJoin and gauge the performance benefits of the operator.

Query 13 involves the *customer* and *orders* tables. The schema of the *customer* and *orders* tables is shown in Table 4.1.

CUSTOMER	
C_CUSTKEY	INTEGER NOT NULL
C_NAME	VARCHAR(25) NOT NULL
C_ADDRESS	VARCHAR(40) NOT NULL
C_NATIONKEY	INTEGER NOT NULL
C_PHONE	CHAR(15) NOT NULL
C_ACCTBAL	DECIMAL(15,2) NOT NULL
C_MKTSEGMENT	CHAR(10) NOT NULL
C_COMMENT	VARCHAR(117) NOT NULL

ORDERS	
O_ORDERKEY	INTEGER NOT NULL
O_CUSTKEY	INTEGER NOT NULL
O_ORDERSTATUS	CHAR(1) NOT NULL
O_TOTALPRICE	DECIMAL(15,2) NOT NULL
O_ORDERDATE	DATE NOT NULL
O_ORDERPRIORITY	CHAR(15) NOT NULL
O_CLERK	CHAR(15) NOT NULL
O_SHIPPRIORITY	INTEGER NOT NULL
O_COMMENT	VARCHAR(79) NOT NULL

Table 4.1: Customer and Orders tables' schema

The result of Query 13 is the distribution of count of customers by the number of orders they have made. The SQL for Query 13 (earlier shown in section 2.1) is shown in Figure 4.1. The corresponding AQL for Query 13 is shown in Figure 4.2.

```
select
    c_count, count(*) as custdist
from
    (
        select
            c_custkey, count(o_orderkey)
        from
            customer left outer join orders on
                c_custkey = o_custkey
                and o_comment not like '%special%requests%'
        group by
            c_custkey
    ) as c_orders (c_custkey, c_count)
group by
    c_count
order by
    custdist desc, c_count desc;
```

Figure 4.1: TPC-H Query 13 in SQL

```
for $gco in (
    for $co in (
        for $c in dataset('Customer')
            return {
                "c_custkey": $c.c_custkey,
                "o_orderkey_count": count(
                        for $o in dataset('Orders')
                        where  $c.c_custkey = $o.o_custkey
                        and not(like($o.o_comment,'%special%requests%'))
                        return $o.o_orderkey)
                }
        )
        /*+ hash */
        group by $c_custkey := $co.c_custkey with $co
        return {
            "c_custkey": $c_custkey,
            "c_count": sum(for $i in $co return $i.o_orderkey_count)
            }
        )
    /*+ hash */
    group by $c_count := $gco.c_count with $gco
    let $custdist := count($gco)
    order by $custdist desc, $c_count desc
    return {"c_count": $c_count, "custdist": $custdist}
```

Figure 4.2: TPC-H Query 13 in AQL

The experiments were run on TPC-H [6] data sets of different sizes, ranging from scale factors of 0.5 to 1000. Since we are concerned with only the *customer* and *orders* tables, these scale factors amounted to datasets ranging in size from 93MB (for SF=0.5) to 194GB (for SF=1000). The time taken to load the datasets into the storage engine is not part of the results.

Query 13 was run for both the GroupJoin plan and the Join+Group plan (by using the query hint described in section 3.2.3) repeatedly using a randomizing script. The results shown in the next section are query execution times averaged over 5 runs. Though the output write time is not included in the results, we found that it was inconsequential even at the smallest scale factor.

4.3 Results

This section presents and analyzes the resulting query plans for Query 13 and also the query execution times for both the plans.

4.3.1 Optimized Query Plans

The detailed Join+Group query plan generated by ASTERIX is shown in Figure 4.3.

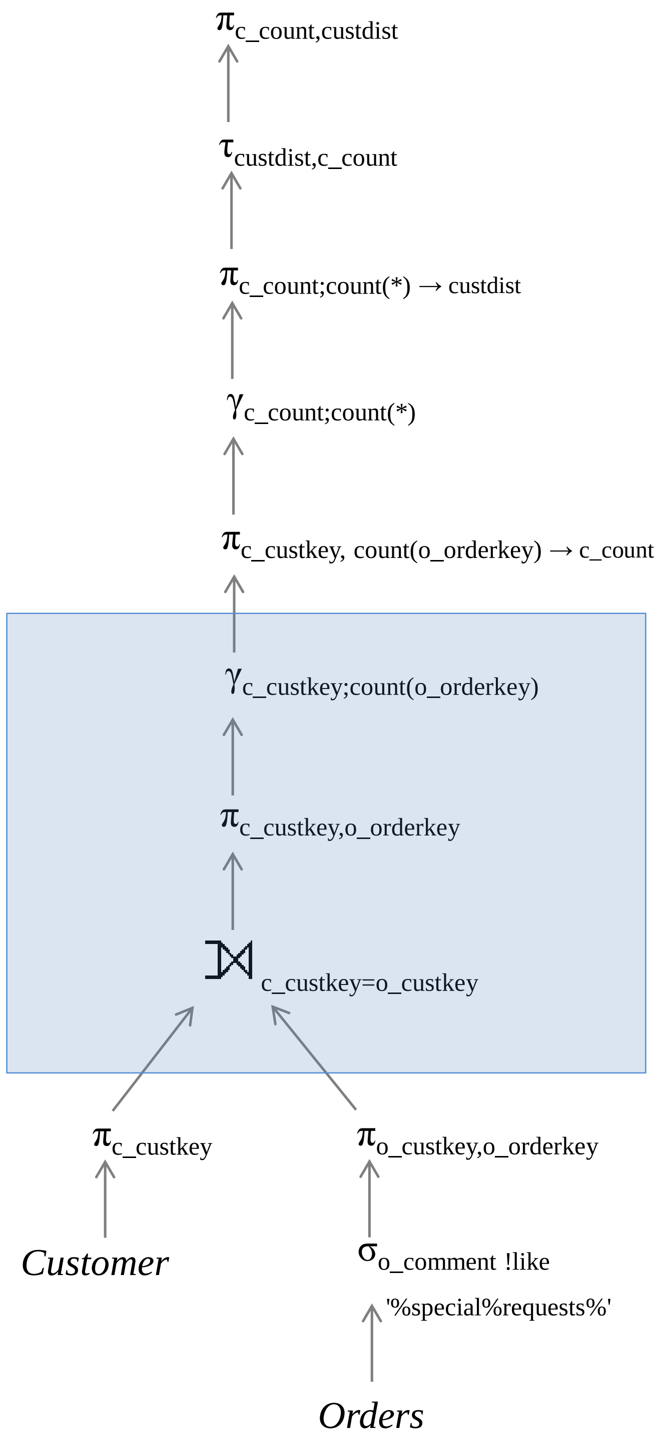

Figure 4.3: Complete ASTERIX plan for Query 13 with Join+Group

Similarly, the GroupJoin query plan is shown in Figure 4.4. The highlighted regions show the difference between the two plans, i.e., the replacement of Join+Group with a GroupJoin.

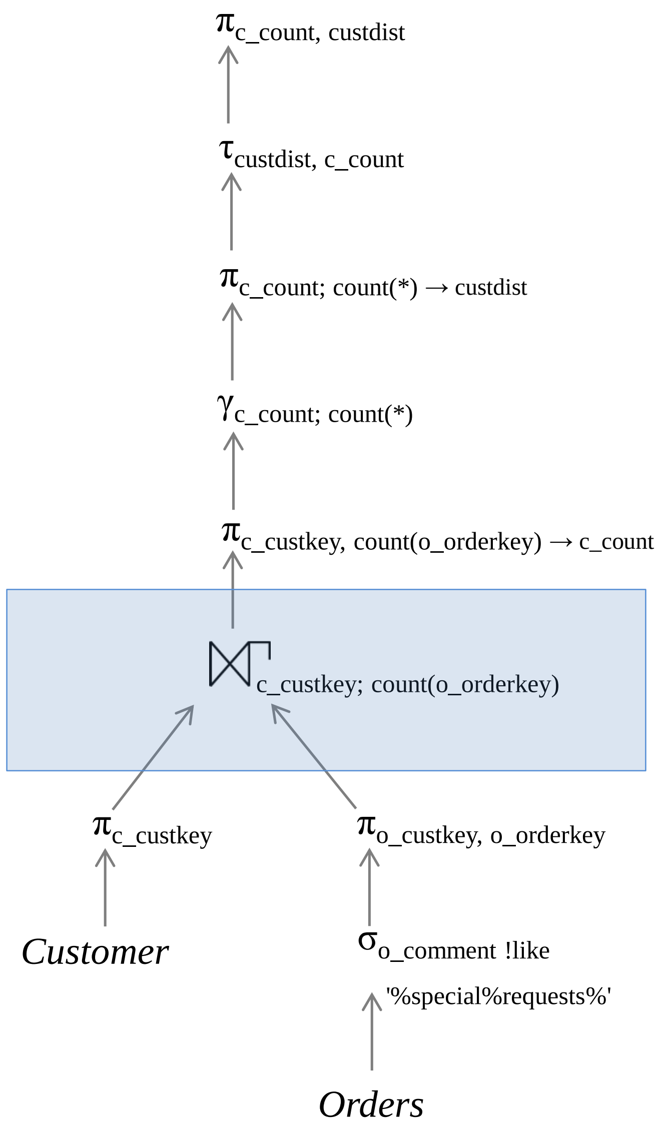
π c_count, custdist
τ custdist, c_count
π c_count; count(*) → custdist
γ c_count; count(*)
π c_custkey, count(o_orderkey) → c_count
c_custkey; count(o_orderkey)
π c_custkey
π o_custkey, o_orderkey
Customer
σ o_comment !like '%special%requests%'
Orders

Figure 4.4: Complete ASTERIX plan for Query 13 with GroupJoin

4.3.2 Run times

Table 4.2 lists the results of the experiments run with Query 13. The Improvement column is the ratio of query run time with Join+Group to that with GroupJoin.

TPC-H SCALE		RUN TIME(s)		IMPROV.
FACTOR	SIZE	JOIN+GROUP PLAN	GROUPJOIN PLAN	
0.5	93MB	2.244	1.934	1.160
1	188MB	2.836	2.390	1.187
5	946MB	8.029	6.712	1.196
10	1.85GB	14.586	12.122	1.203
25	4.62GB	34.221	28.074	1.219
50	9.15GB	89.073	72.556	1.228
100	18.3GB	181.524	147.719	1.229
250	46.1GB	454.667	371.333	1.224
500	92GB	895.000	732.000	1.223
1000	194GB	1795.000	1459.000	1.230

Table 4.2: Query 13 execution times for different TPC-H dataset scales

The results show Query 13 execution times for data sets ranging from 93MB to 194GB. The performance improvement when using GroupJoin was found to be up to 1.23x. This improvement was mainly due to the omission of creating the hash table in the Grouping operator in the Join+Group plan. Operator pipelining ensures that, for the dataset sizes tested here, there is no spillage of Join results to disk in the case of the Join+Group plan. Join results are fed to the Grouping operator as they are created, so the GroupJoin does not provide an I/O cost savings over the Join+Group plan here.

A plot of Improvement Factor vs. TPC-H data set scale is shown in Figure 4.5. We can see that after initially increasing, the Improvement Factor becomes steady at about 1.2x.

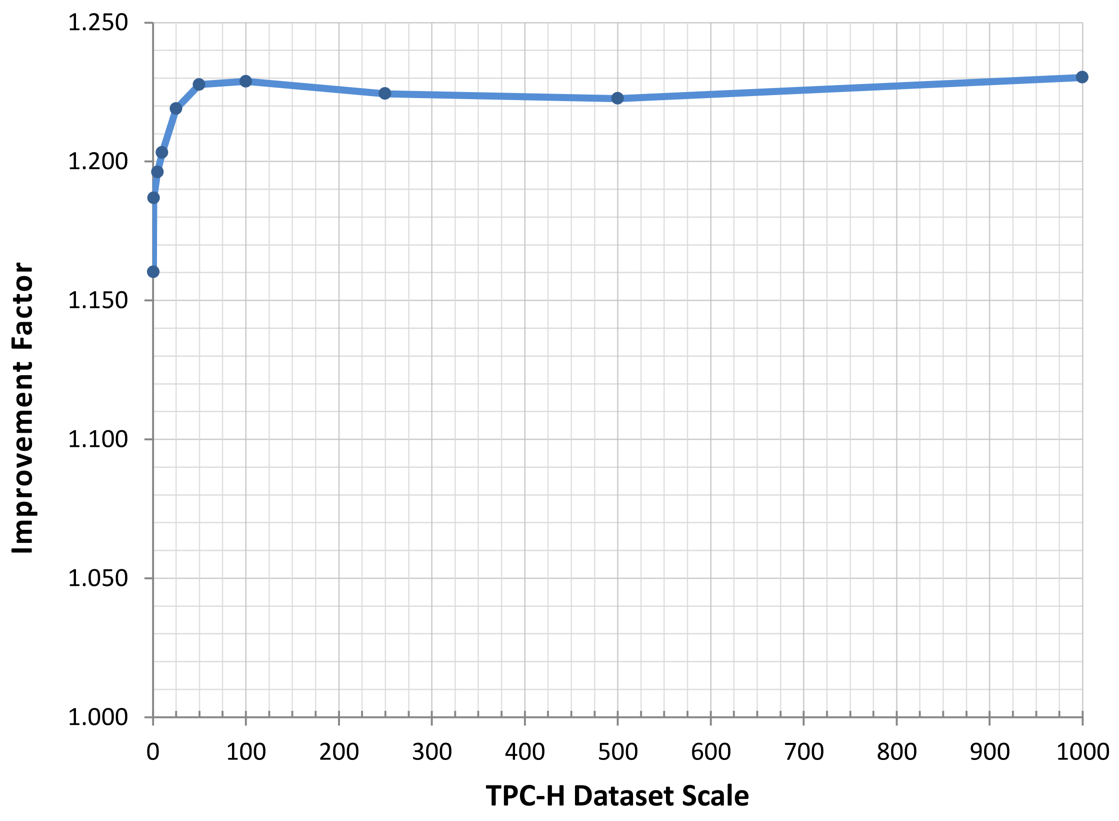

Figure 4.5: Improvement Factor vs. TPC-H Dataset Scale

4.3.3 Conclusion

This chapter was devoted to a performance evaluation and comparison of the GroupJoin and Join+Group plans. The improvement due to the use of GroupJoin was found to be about 1.2x in experiments involving Query 13 from TPC-H. It is interesting to note that the improvement due to the use of GroupJoin does not match the 3x improvement reported in [2]. This is because performance in our context is driven by disk I/O cost, and data is effectively pipelined from Join to Grouping in the Join+Group plan. Thus, GroupJoin did not offer an I/O savings and we did not see such dramatic performance benefits in our experiments.

Chapter 5

Conclusion

In this thesis, we reported on an effort to implement the GroupJoin operator in the ASTERIX BDMS. We showed the use and value of the operator and discussed the conditions for GroupJoin equivalence. The GroupJoin algorithm was described for both In-Memory and Hybrd Hash GroupJoin operators. The ASTERIX software stack was briefly discussed, followed by GroupJoin implementation details specific to ASTERIX. Finally, a performance evaluation of the GroupJoin based on a query from TPC-H was described along with the results. The GroupJoin was found to improve query performance for this query by a factor of about 1.2.

Bibliography

[1] G. von Bultzingsloewen. Optimizing sql queries for parallel execution. In *ACM SIGMOD Record, 18:17-22*, December 1989.

[2] G. Moerkotte and T. Neumann. Accelerating queries with groupby and join by groupjoin. In *Proceedings of the VLDB Endowment, Vol. 4, No. 11*, August 2011.

[3] S. Alsubaiee, A. Behm, R. Grover, R. Vernica, V. R. Borkar, M. J. Carey, and C. Li. Asterix: Scalable warehouse-style web data integration. In *IIWeb*, May 2012.

[4] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, 2009.

[5] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall, 2002.

[6] *The TPC-H Benchmark*. `http://www.tpc.org/tpch`.

[7] S. Cluet and G. Moerkotte. Efficient evaluation of aggregates on bulk types. In *Proc. Int. Workshop on Database Programming Languages*, 1995.

[8] V. R. Borkar, M. J. Carey, and C. Li. Inside "big data management": Ogres, onions, or parfaits? In *EDBT/ICDT 2012 Joint Conference*, March 2012.

[9] V. R. Borkar, M. J. 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*, 2011.

[10] V. R. Borkar. A toolkit for the efficient processing of big data on large clusters. In *VLDB*, August 2012.

[11] A. Behm, V. R. Borkar, M. J. Carey, R. Grover, , C. Li, N. Onose, R. Vernica, A. Deutsch, Y. Papakonstantinou, and V. J. Tsotras. Asterix: towards a scalable, semistructured data platform for evolving-world models. In *Distributed Parallel Databases*, March.

[12] *JSON*. `http://www.json.org/`.

[13] R. G. G. Cattell. *The Object Database Standard: ODMG 2.0*. Morgan Kauffman.

[14] *Object database management systems*. `http://www.odbms.org/odmg/`.

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

[16] D. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. Wood. Implementation techniques for main memory database systems. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, 1984.

Appendices

A Aggregate Decomposition

Let A be a decomposable Aggregate function. Also, Let A$_1$ and A$_2$ be Aggregate functions such that $A = A_2(A_1)$. That is, A is decomposed into the two Aggregate functions A$_1$ and A$_2$. The following table lists decomposable Aggregate functions and their decompositions A$_1$ and A$_2$.

A	A$_1$	A$_2$
sum	sum	sum
count	count	sum
avg	sum, count	sum, sum
min	min	min
max	max	max

It should be noted that as long as A$_1$ and A$_2$ are decomposable, the Aggregate function A can be expressed in terms of any number of decomposotions. For example, the *count* function can be decomposed as

$$sum(sum(sum(sum(count))))$$

without loss of accuracy.