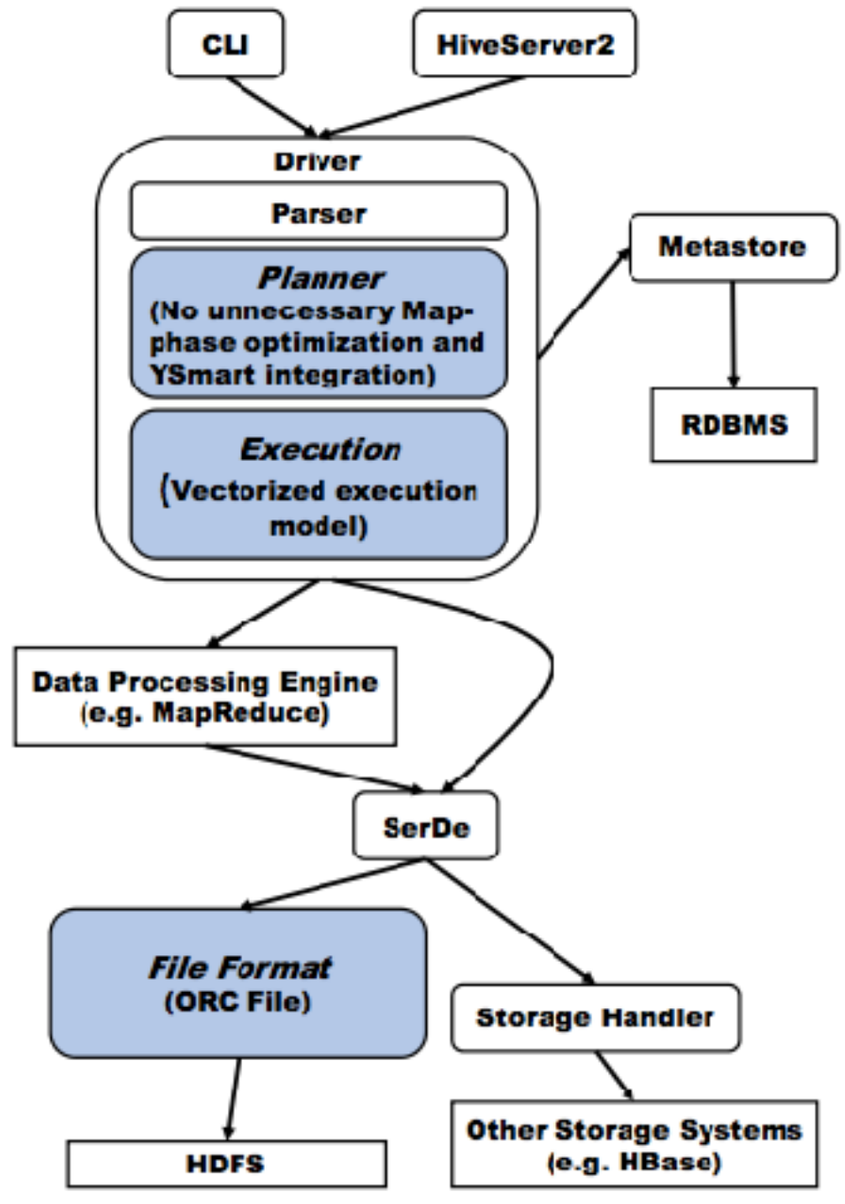
Advancements in Apache Hive

Hive Architecture

- Command Line Interface and HiveServer2
- Driver to plan query and handle execution
- Processing data in compute engine
- Store the data into files in HDFS



Major Advancements

- File format
- Query Planning
- Query Execution

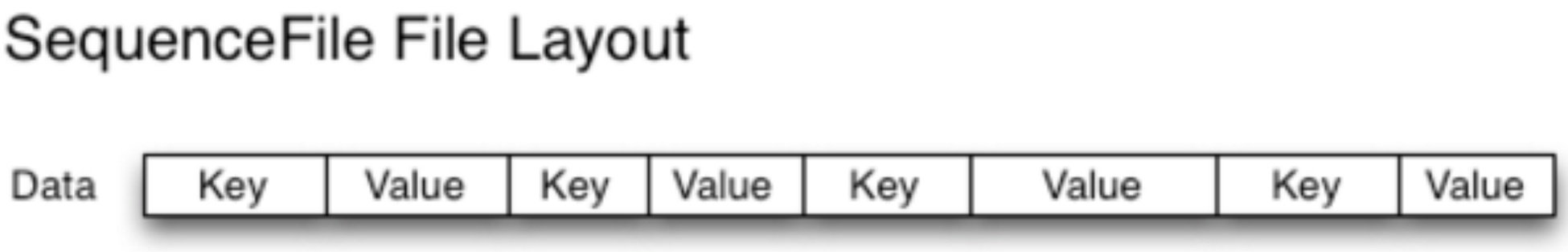
File Format

- Hadoop TextFile and SequenceFile
- Record Columnar File
- Optimised Record Columnar File

File Format

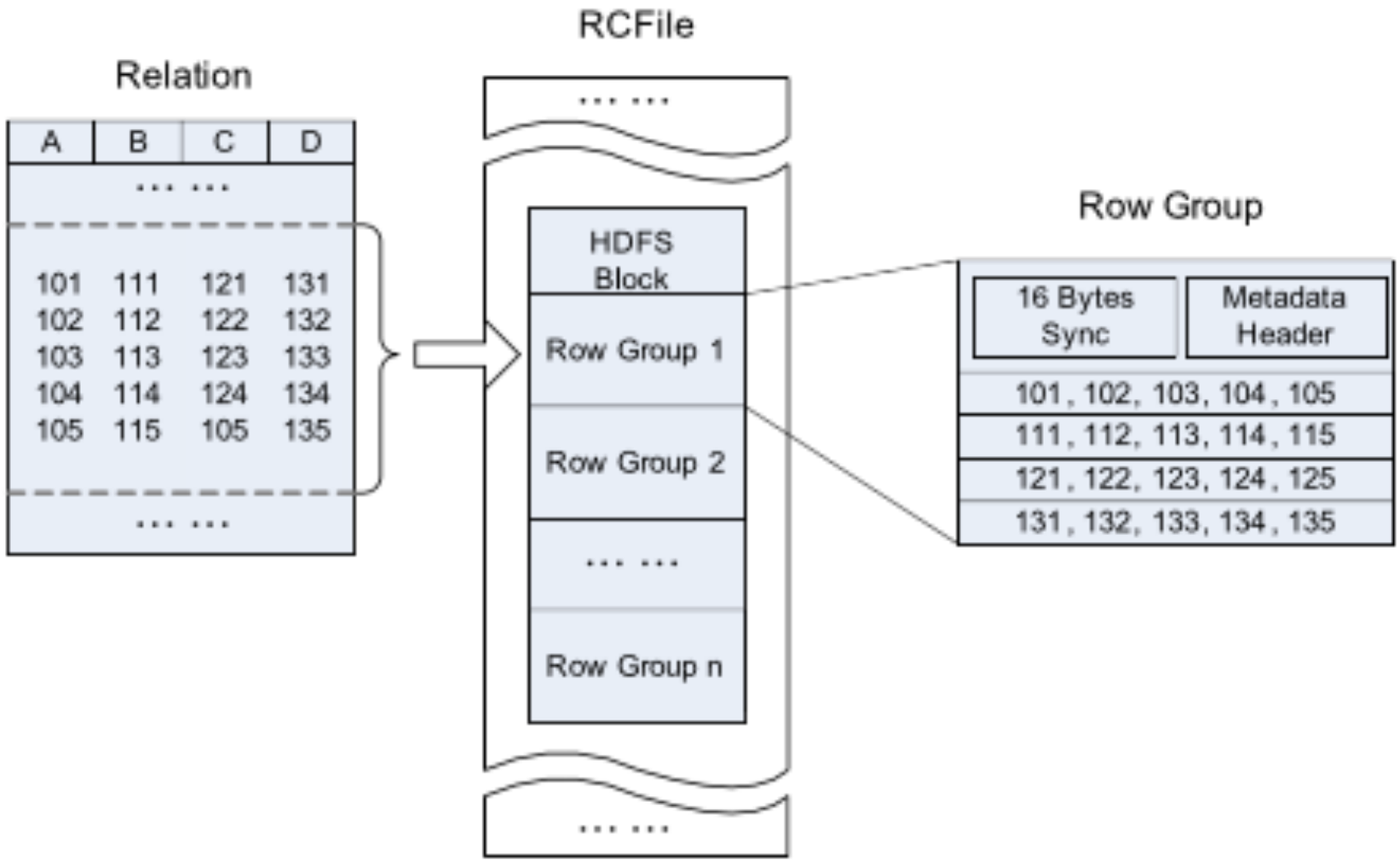
TextFile and SequenceFile:

- TextFile contains plain text data
- SequenceFile is to deal with small file problem:



File Format

Record Columnar File:



He, Yongqiang, et al. "RCFile: A fast and space-efficient data placement structure in MapReduce-based warehouse systems." Data Engineering (ICDE), 2011 IEEE 27th International Conference on. IEEE, 2011.

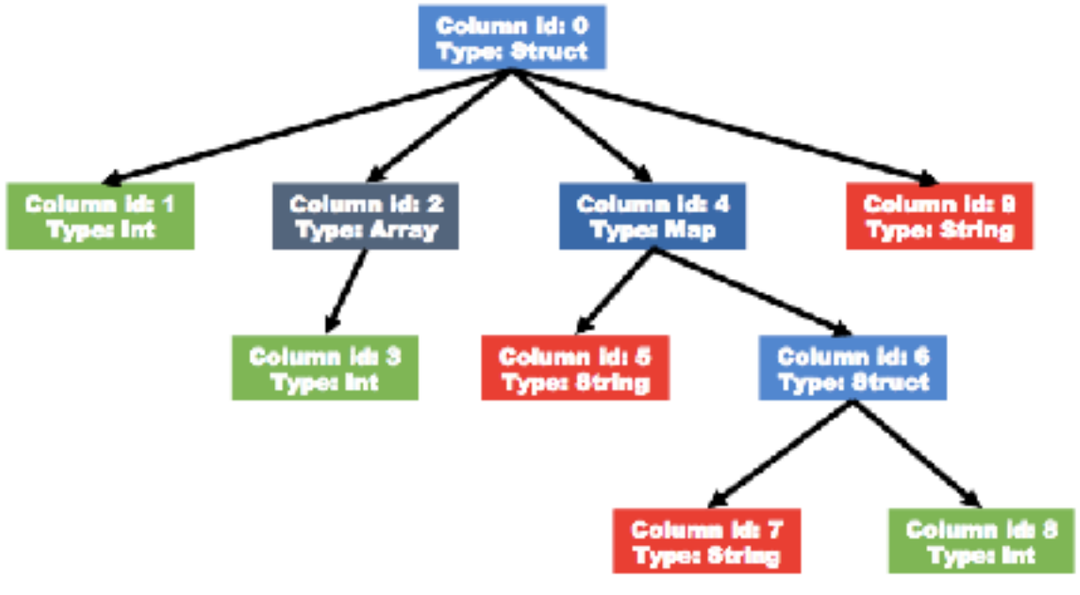
File Format

Optimised Record Columnar File:

- Table placement method
- Indexes
- Compression
- Memory Manager

```
CREATE TABLE tbl (
  col1 Int,
  col2 Array<Int>,
  col4 Map<String,
           Struct<col7:String,
                  col8:Int>>,
  col9 String
)
```

(a) The schema of the table

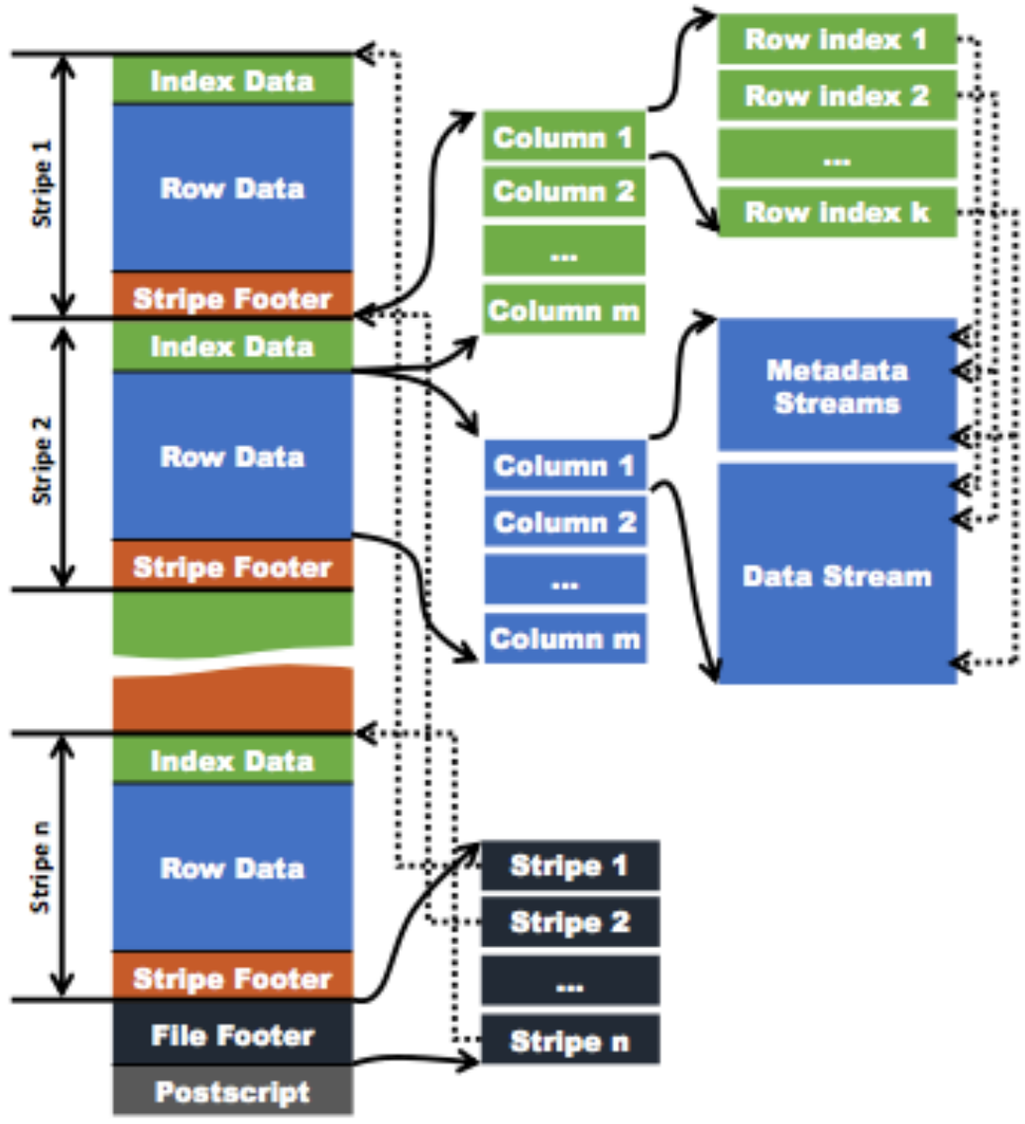


(b) The column tree after column decomposition.

File Format

Optimised Record Columnar File:

- Table placement method
- Indexes
- Compression
- Memory Manager



File Format

Optimised Record Columnar File:

- Table placement method
- Indexes
- Compression
- Memory Manager

File Format

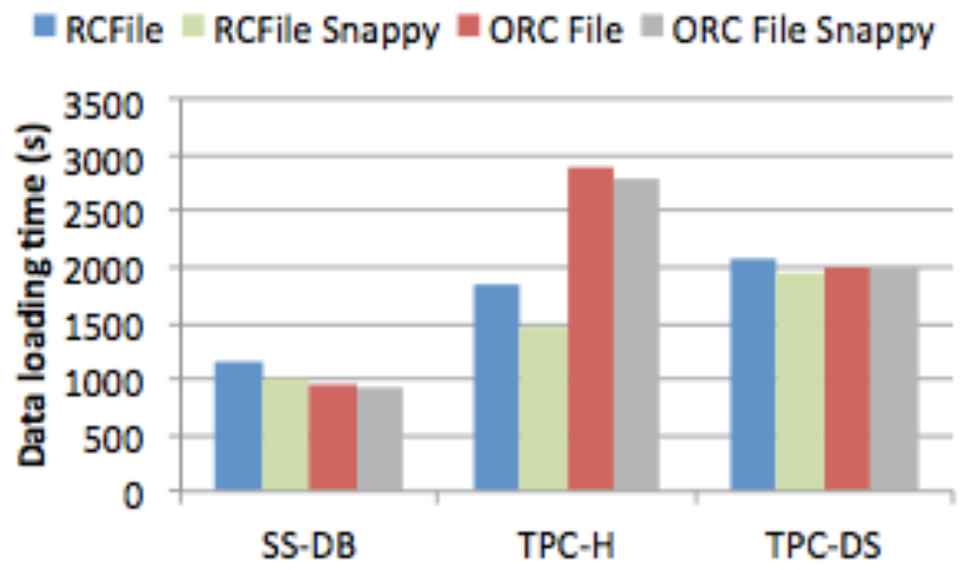
Optimised Record Columnar File:

- Data type aware for type-specific data coding
- Different indexes to skip data reading
- Decompose complex structure
- A larger default stripe size
- Memory management

File Format

Evaluation:

	SS-DB	TPC-H	TPC-DS
Text	248.35	323.84	279.87
RCFile	128.23	269.00	159.69
RCFile Snappy	55.15	118.33	105.28
ORC File	53.51	168.96	102.24
ORC File Snappy	39.20	86.67	94.05

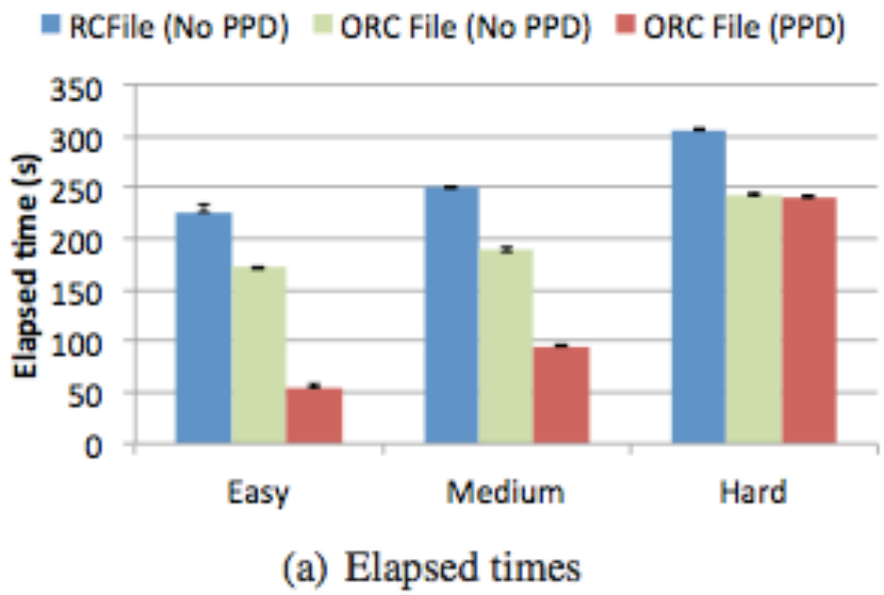


Conclusion: Size of datasets is small in ORC File, and ORC File is competitive in terms of data loading time.

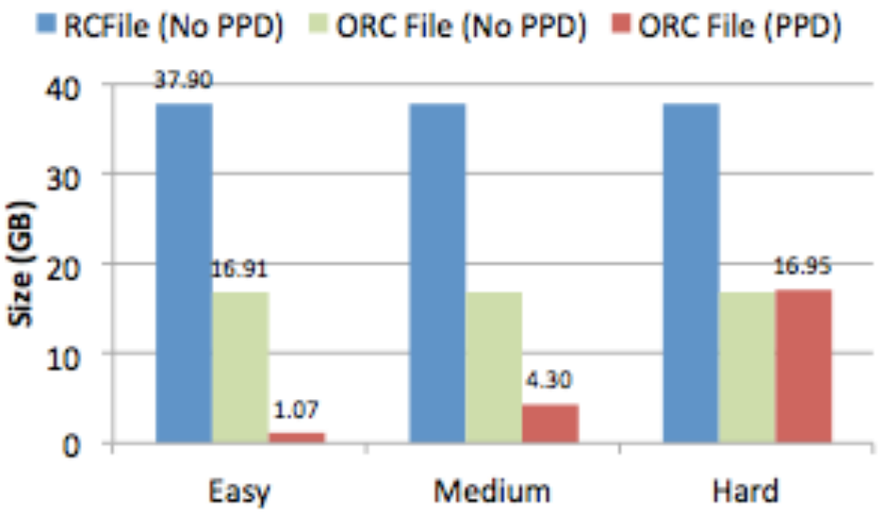
File Format

Evaluation:

```
SELECT SUM(v1), COUNT(*) FROM cycle
WHERE x BETWEEN 0 and var AND
      y BETWEEN 0 and var;
```



(a) Elapsed times



(b) Amounts of data read from HDFS

Conclusion: The ORC File is more efficient in the actual data query.

Query Planning

- Unnecessary Map phase
- Unnecessary data loading
- Unnecessary data re-partitioning

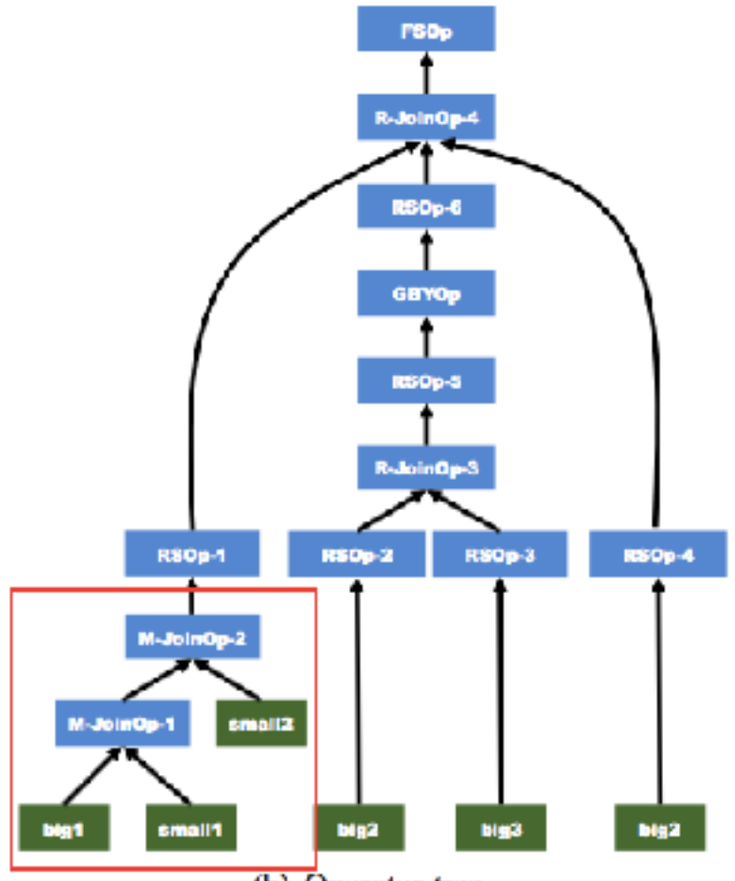
Query Planning

Eliminating Unnecessary Map phase:

- Map join can reduce the shuffle phase cost
- Map-only job for every map join
- Combine multiple map joins into one jobs

```
SELECT big1.key, small1.value1, small2.value1,
       big2.value1, sq1.total
FROM big1
JOIN small1 ON (big1.sKey1 = small1.key)
JOIN small2 ON (big1.sKey2 = small2.key)
JOIN (SELECT key,
             avg(big3.value1) AS avg,
             sum(big3.value2) AS total
      FROM big2 JOIN big3 ON (big2.key = big3.key)
      GROUP BY big2.key) sq1 ON (big1.key = sq1.key)
JOIN big2 ON (sq1.key = big2.key)
WHERE big2.value1 > sq1.avg;
```

(a) Query



(b) Operator tree

Query Planning

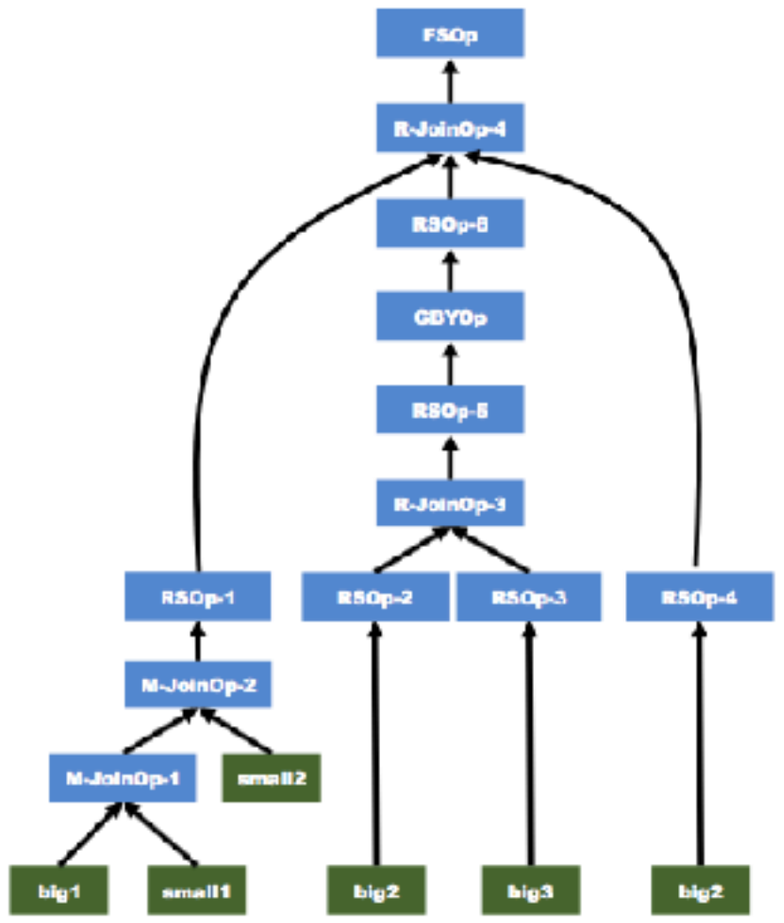
Correlation Optimiser:

- Input correlation: a table is used by multiple operations

- Job flow correlation: A major operator depends on another major operator

```
SELECT big1.key, small1.value1, small2.value1,
       big2.value1, sq1.total
FROM big1
JOIN small1 ON (big1.sKey1 = small1.key)
JOIN small2 ON (big1.sKey2 = small2.key)
JOIN (SELECT key,
             avg(big3.value1) AS avg,
             sum(big3.value2) AS total
      FROM big2 JOIN big3 ON (big2.key = big3.key)
      GROUP BY big2.key) sq1 ON (big1.key = sq1.key)
JOIN big2 ON (sq1.key = big2.key)
WHERE big2.value1 > sq1.avg;
```

(a) Query



(b) Operator tree

Query Planning

Correlation Detection:

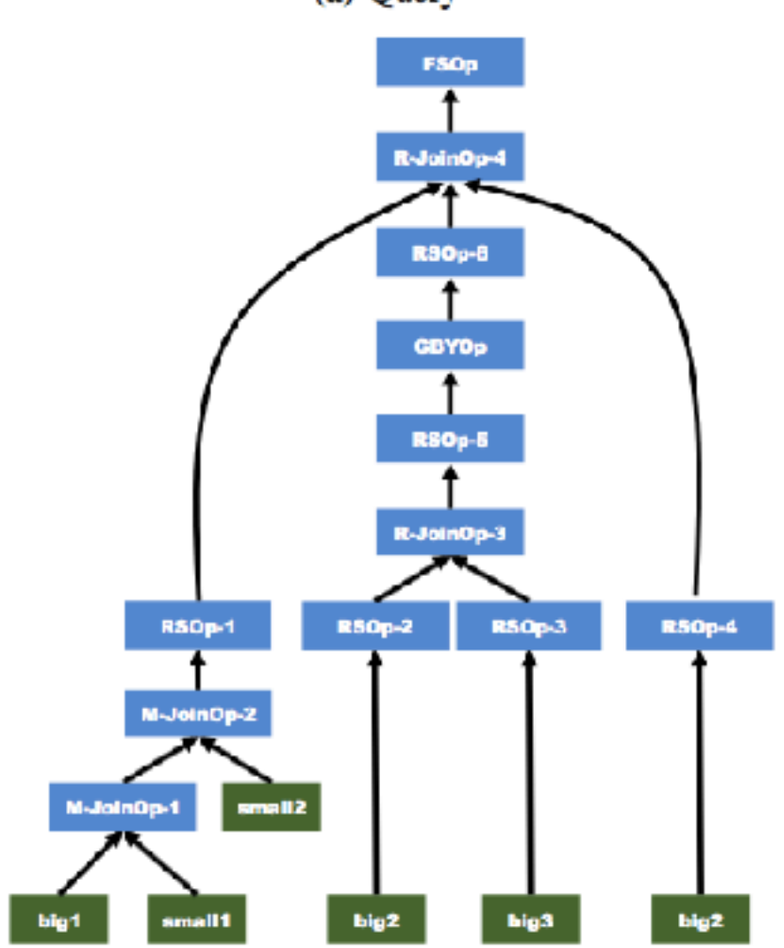
1. Emitted rows from these two RSOps are sorted in the same way

2. Emitted rows from these two RSOps are partitioned in the same way

3. RSOps do not have any conflict on the number reducers.

```
SELECT big1.key, small1.value1, small2.value1,
       big2.value1, sq1.total
FROM big1
JOIN small1 ON (big1.sKey1 = small1.key)
JOIN small2 ON (big1.sKey2 = small2.key)
JOIN (SELECT key,
             avg(big3.value1) AS avg,
             sum(big3.value2) AS total
      FROM big2 JOIN big3 ON (big2.key = big3.key)
      GROUP BY big2.key) sq1 ON (big1.key = sq1.key)
JOIN big2 ON (sq1.key = big2.key)
WHERE big2.value1 > sq1.avg;
```

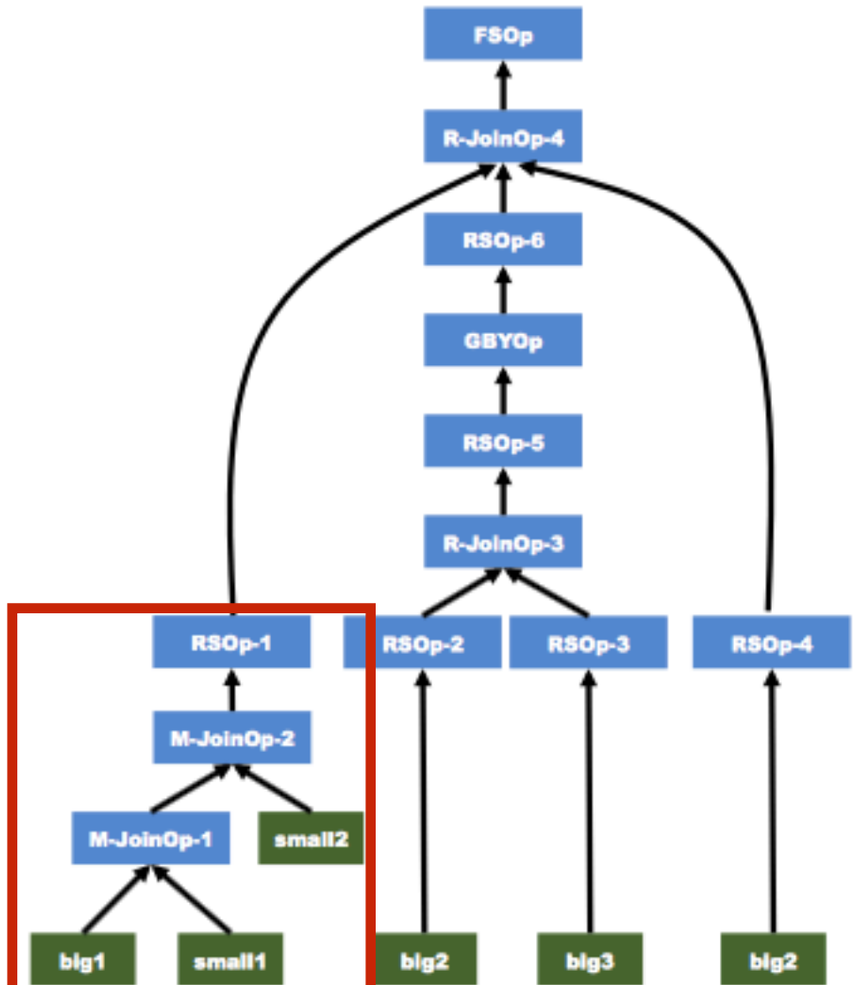
(a) Query



(b) Operator tree

Query Planning

```
SELECT big1.key, small1.value1, small2.value1,
       big2.value1, sq1.total
FROM big1
JOIN small1 ON (big1.sKey1 = small1.key)
JOIN small2 ON (big1.sKey2 = small2.key)
JOIN (SELECT key,
             avg(big3.value1) AS avg,
             sum(big3.value2) AS total
      FROM big2 JOIN big3 ON (big2.key = big3.key)
      GROUP BY big2.key) sq1 ON (big1.key = sq1.key)
JOIN big2 ON (sq1.key = big2.key)
WHERE big2.value1 > sq1.avg;
```

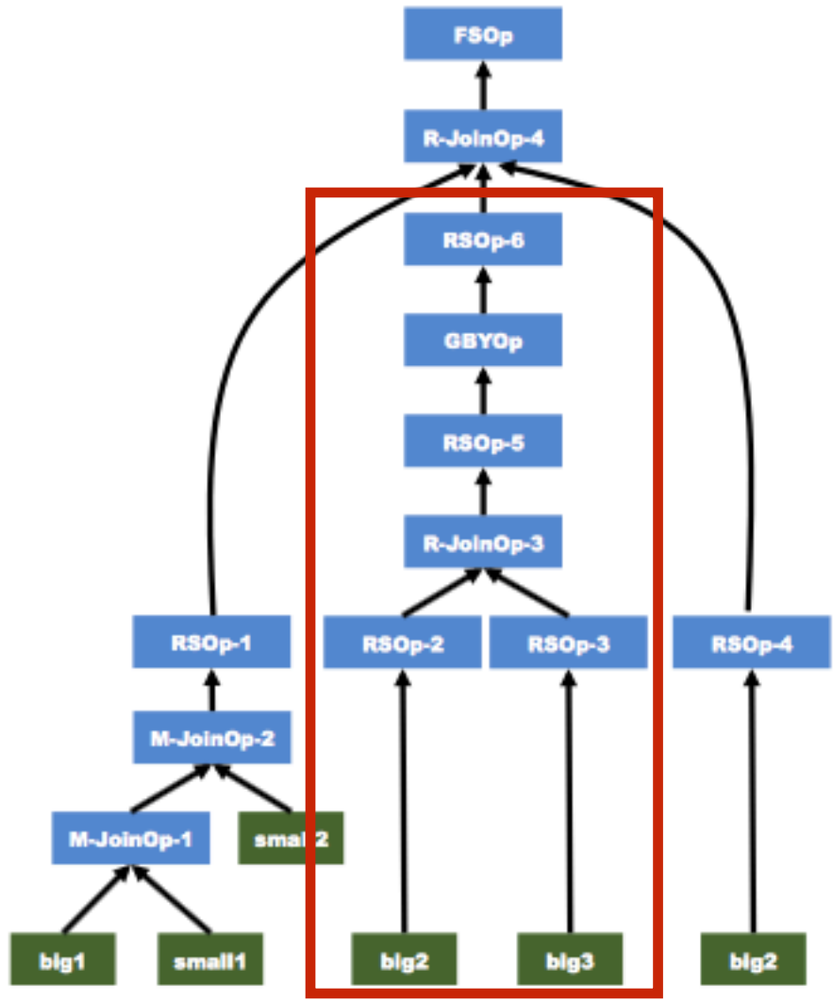


(b) Operator tree

Figure 4: The running example used in Section 5. For an arrow connecting two operators, it starts from the parent operator and ends at the child operator.

Query Planning

```
SELECT big1.key, small1.value1, small2.value1,
       big2.value1, sq1.total
FROM big1
JOIN small1 ON (big1.sKey1 = small1.key)
JOIN small2 ON (big1.sKey2 = small2.key)
JOIN (SELECT key,
             avg(big3.value1) AS avg,
             sum(big3.value2) AS total
      FROM big2 JOIN big3 ON (big2.key = big3.key)
      GROUP BY big2.key) sq1 ON (big1.key = sq1.key)
JOIN big2 ON (sq1.key = big2.key)
WHERE big2.value1 > sq1.avg;
```

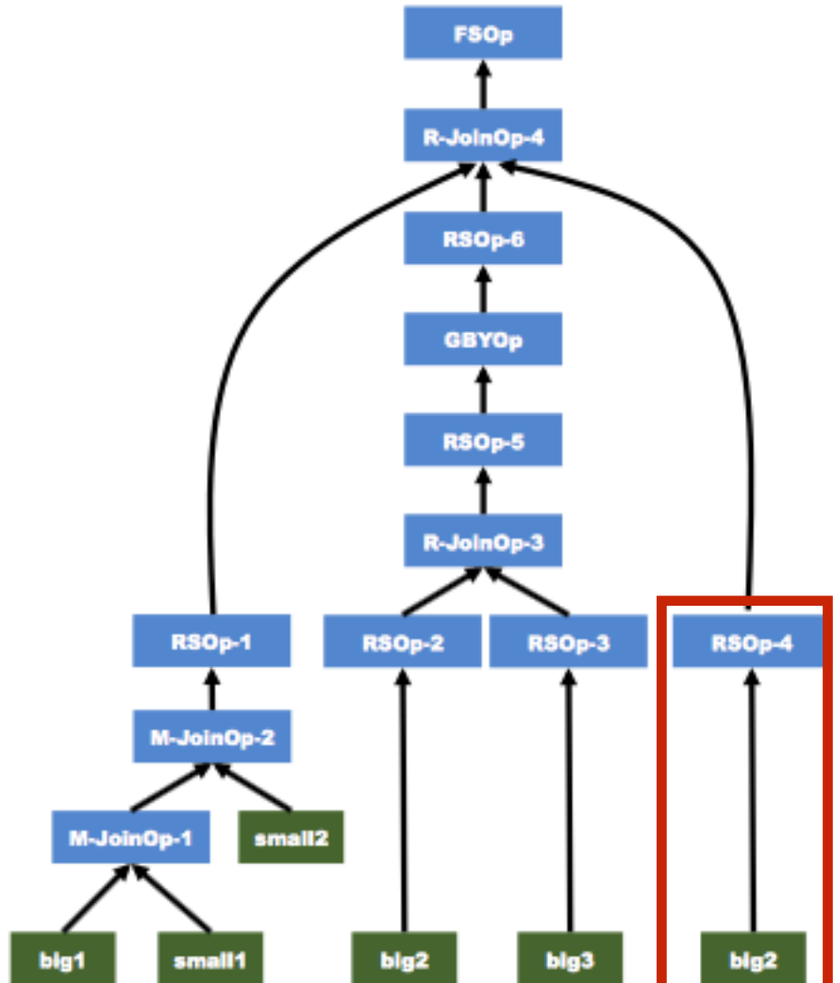


(b) Operator tree

Figure 4: The running example used in Section 5. For an arrow connecting two operators, it starts from the parent operator and ends at the child operator.

Query Planning

```
SELECT big1.key, small1.value1, small2.value1,
       big2.value1, sq1.total
FROM big1
JOIN small1 ON (big1.sKey1 = small1.key)
JOIN small2 ON (big1.sKey2 = small2.key)
JOIN (SELECT key,
             avg(big3.value1) AS avg,
             sum(big3.value2) AS total
      FROM big2 JOIN big3 ON (big2.key = big3.key)
      GROUP BY big2.key) sq1 ON (big1.key = sq1.key)
JOIN big2 ON (sq1.key = big2.key)
WHERE big2.value1 > sq1.avg;
```

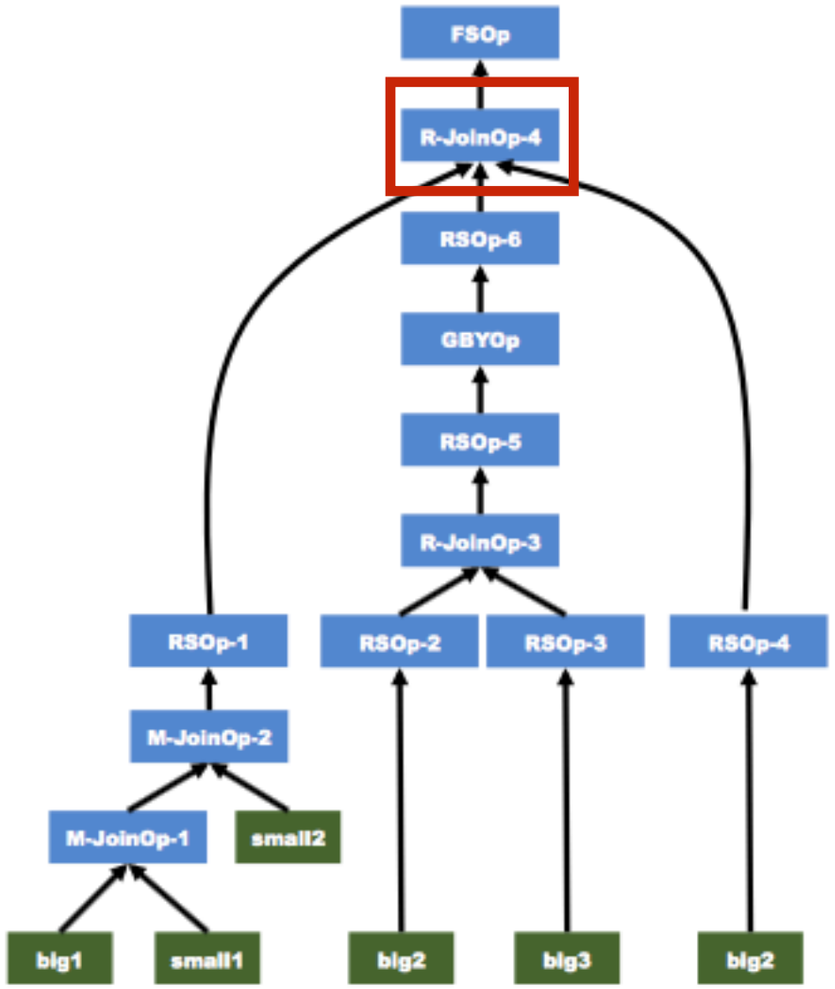


(b) Operator tree

Figure 4: The running example used in Section 5. For an arrow connecting two operators, it starts from the parent operator and ends at the child operator.

Query Planning

```
SELECT big1.key, small1.value1, small2.value1,
       big2.value1, sq1.total
FROM big1
JOIN small1 ON (big1.sKey1 = small1.key)
JOIN small2 ON (big1.sKey2 = small2.key)
JOIN (SELECT key,
             avg(big3.value1) AS avg,
             sum(big3.value2) AS total
      FROM big2 JOIN big3 ON (big2.key = big3.key)
      GROUP BY big2.key) sq1 ON (big1.key = sq1.key)
JOIN big2 ON (sq1.key = big2.key)
WHERE big2.value1 > sq1.avg;
```



(b) Operator tree

Figure 4: The running example used in Section 5. For an arrow connecting two operators, it starts from the parent operator and ends at the child operator.

Query Planning

```
SELECT big1.key, small1.value1, small2.value1,
       big2.value1, sq1.total
FROM big1
JOIN small1 ON (big1.sKey1 = small1.key)
JOIN small2 ON (big1.sKey2 = small2.key)
JOIN (SELECT key,
             avg(big3.value1) AS avg,
             sum(big3.value2) AS total
      FROM big2 JOIN big3 ON (big2.key = big3.key)
      GROUP BY big2.key) sq1 ON (big1.key = sq1.key)
JOIN big2 ON (sq1.key = big2.key)
WHERE big2.value1 > sq1.avg;
```

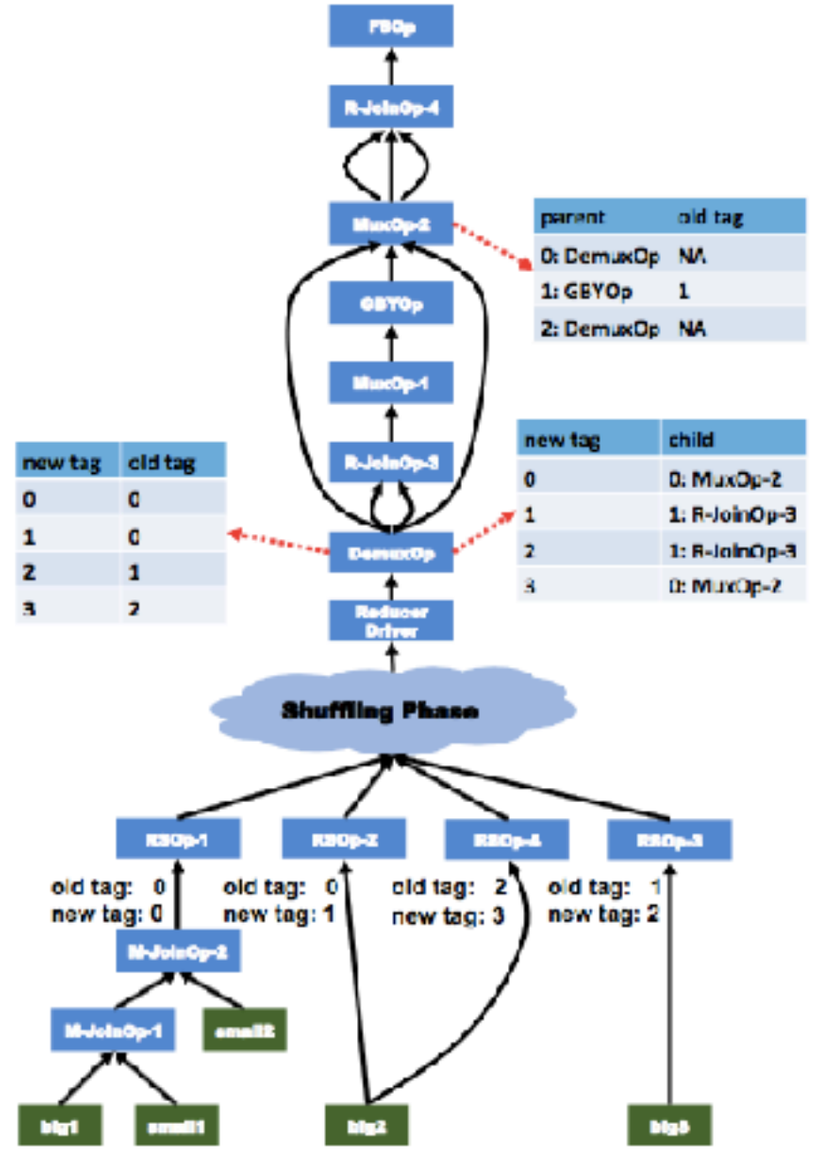
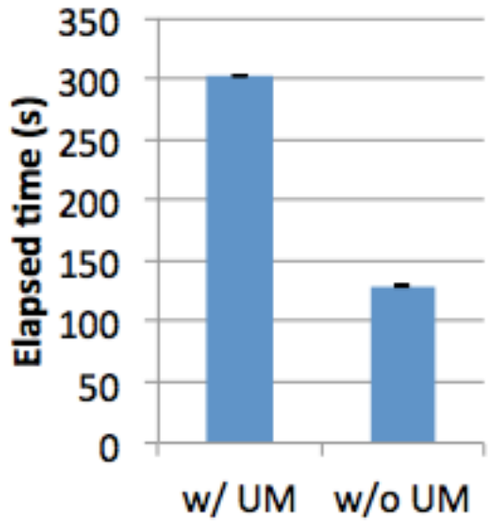


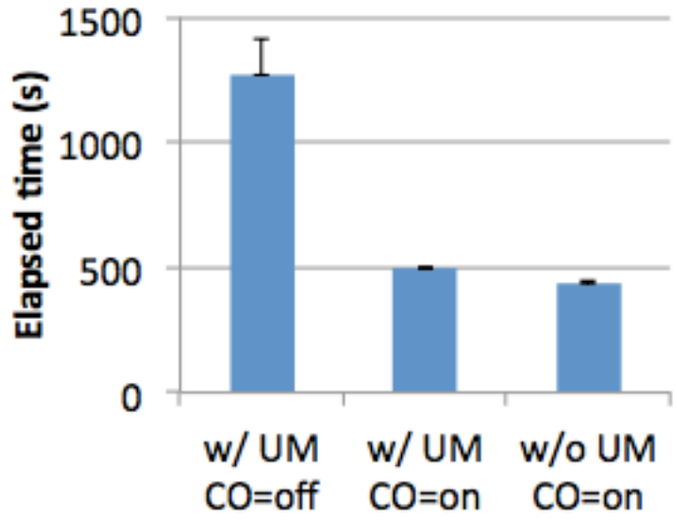
Figure 5: The optimized operator tree of the running example shown in Figure 4.

Query Planning

Evaluation:



(a) TPC-DS query 27



(b) TPC-DS query 95

Conclusion: The elimination of unnecessary map tasks and correlation optimization can improve query performance.

Query Execution

Motivation: Modern CPUs rely on pipelines/parallelism

- Avoid unnecessary branches in the instruction
- Higher data independence among the instructions
- Do not process rows in a one-row-at-a-time manner

Query Execution

- Vectorised Query Execution
- Vectorised Expression
- Vectorised Expression Template
- Vectorised Optimiser
- Vectorised Reader

Query Execution

- One-row-at-a-time: each row traverses the whole operator tree at a time
- Vectorised Query Execution: Apply expression on the entire column vector

```
class LongColumnAddLongScalarExpression {
  int inputColumn;
  int outputColumn;
  long scalar;
  void evaluate(VectorizedRowBatch batch) {
    long [] inVector = ((LongColumnVector)
      batch.columns[inputColumn]).vector;
    long [] outVector = ((LongColumnVector)
      batch.columns[outputColumn]).vector;
    if (batch.selectedInUse) {
      for (int j = 0; j < batch.size; j++) {
        int i = batch.selected[j];
        outVector[i] = inVector[i] + scalar;
      }
    } else {
      for (int i = 0; i < batch.size; i++) {
        outVector[i] = inVector[i] + scalar;
      }
    }
  }
}
```

code snippet of column vector

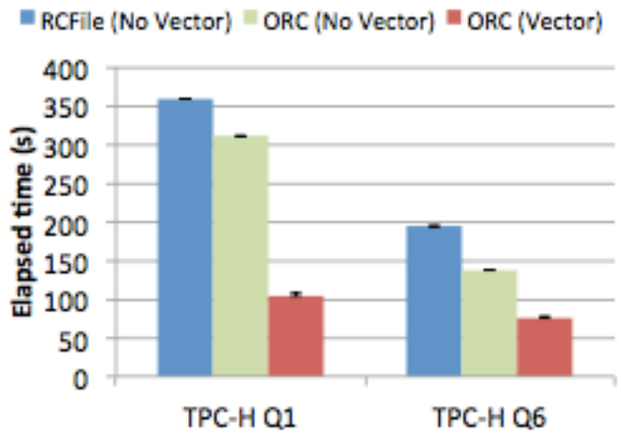
Query Execution

Other things:

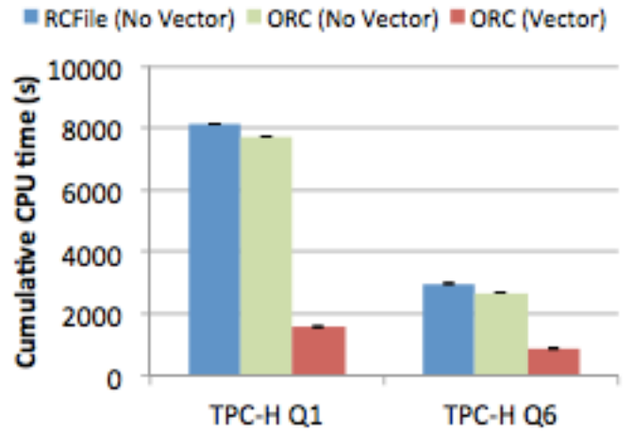
- Specialised vectorised expressions
- Rule-based Vectorisation Optimiser
- Vectorised Reader to provide vectorised row data

Query Planning

Evaluation:



(a) Total elapsed times



(b) Cumulative CPU times

Conclusion: The vectorised query execution can improve query performance.

Summary

- Optimised Record Columnar File
- Query planning updated
- Vectorised query execution