Invisible loading: Access-Driven Data Transfer from Raw Files into Database Systems

Presenter: Hefu Chai

Motivation

- Problems with database systems
  - High “time-to-first-analysis”
  - Large scientific datasets and social networks datasets
  - Non-trivial data preparation
- Advantages of database systems
  - Optimized data layout and query execution plan

Motivation

- Problems with Hadoop
  - Poor cumulative long-term performance
- Advantages of Hadoop
  - Scalable
  - Low “time-to-first” analysis

HadoopDB

Goals

- To achieve low time-to-first analysis of MapReduce jobs over a distributed file system
- To yield the long-term performance benefits of database system

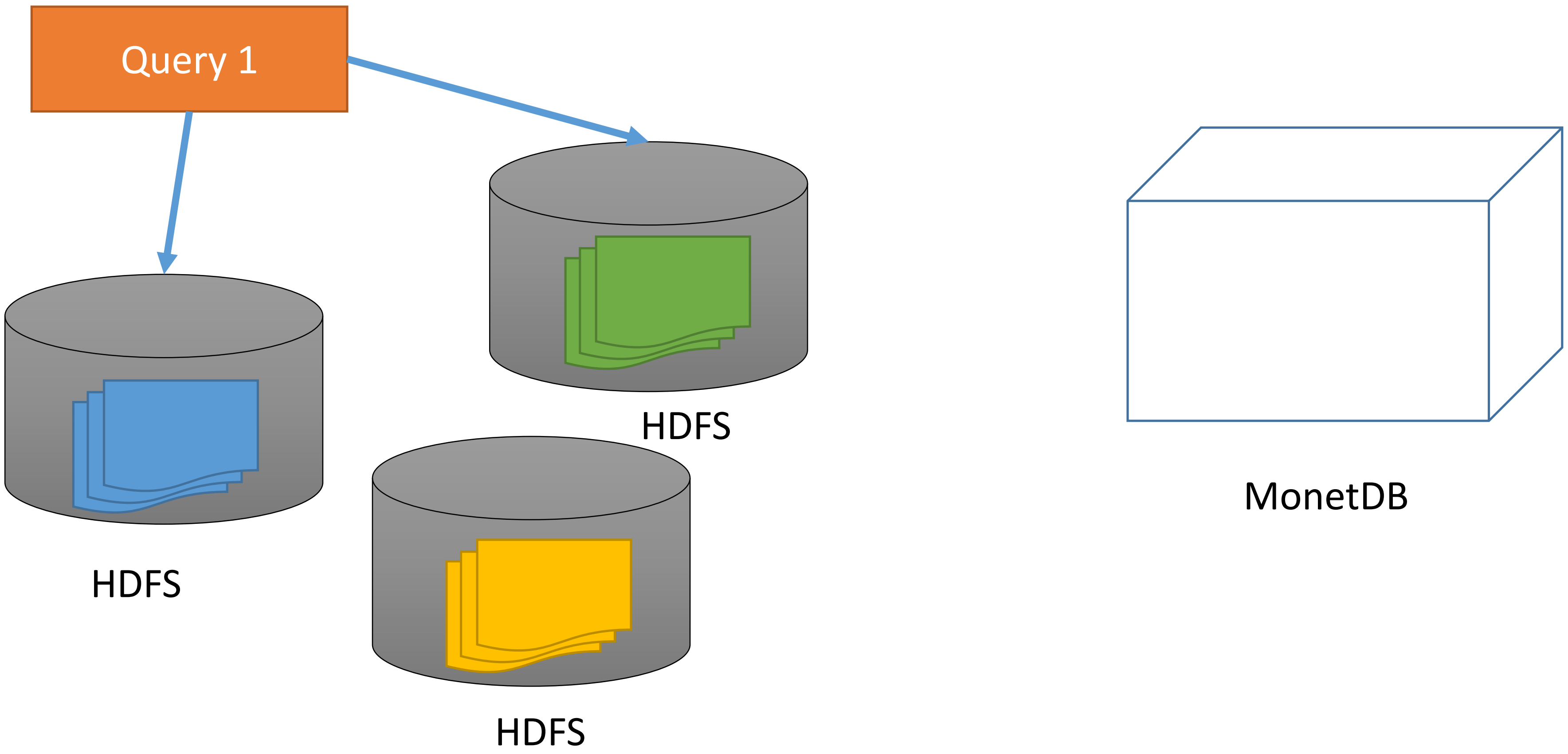
Basic Ideas

- Piggyback on MapReduce jobs
  - Incrementally loading data into databases with almost no marginal cost.
  - Simultaneously processing the data.

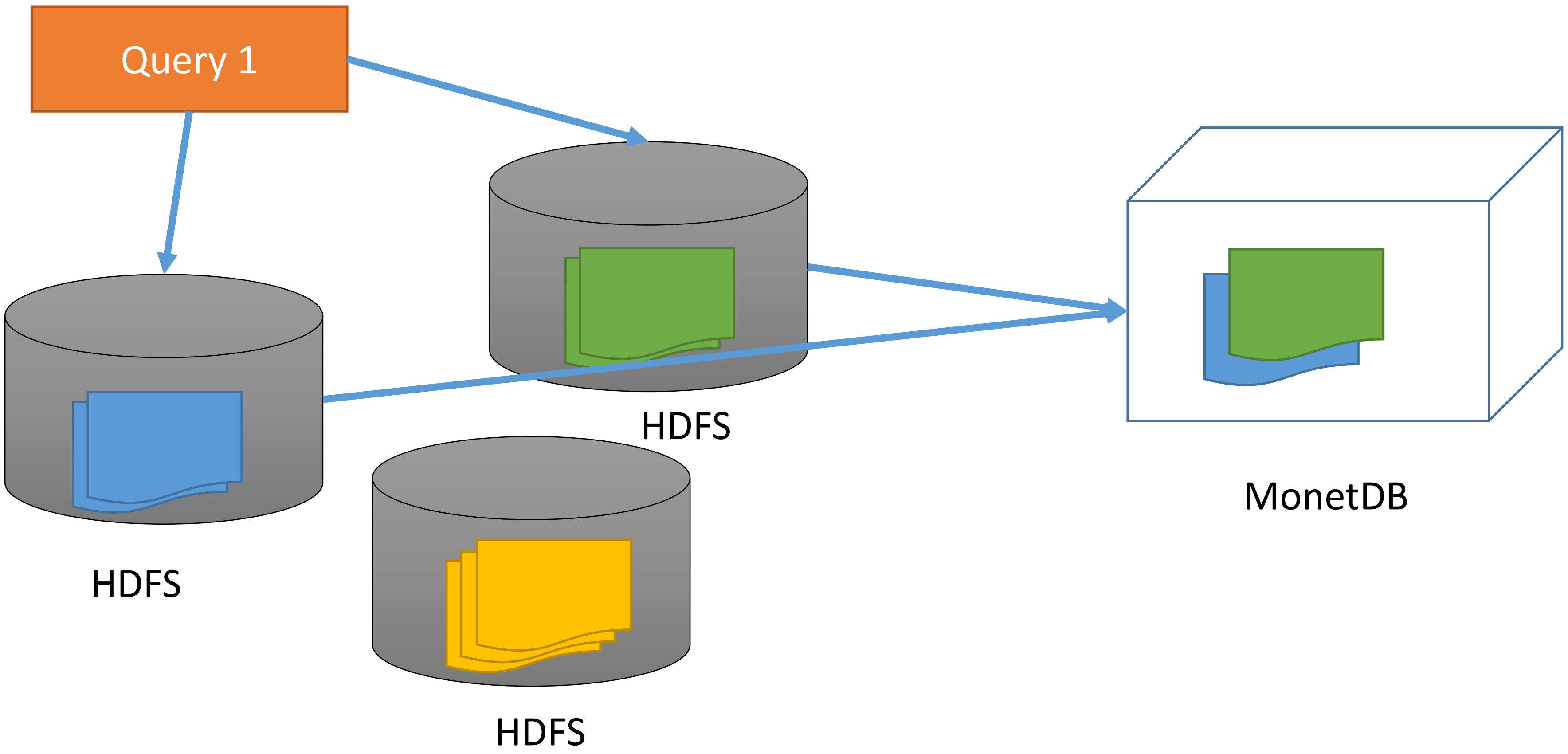
Specific Goal

- Move data from a file system to a database system, with minimal human intervention and human detection (**Invisible**)
  - User should not be forced to specify a complete schema, or database loading operations
  - User should not notice the additional performance overhead of loading work

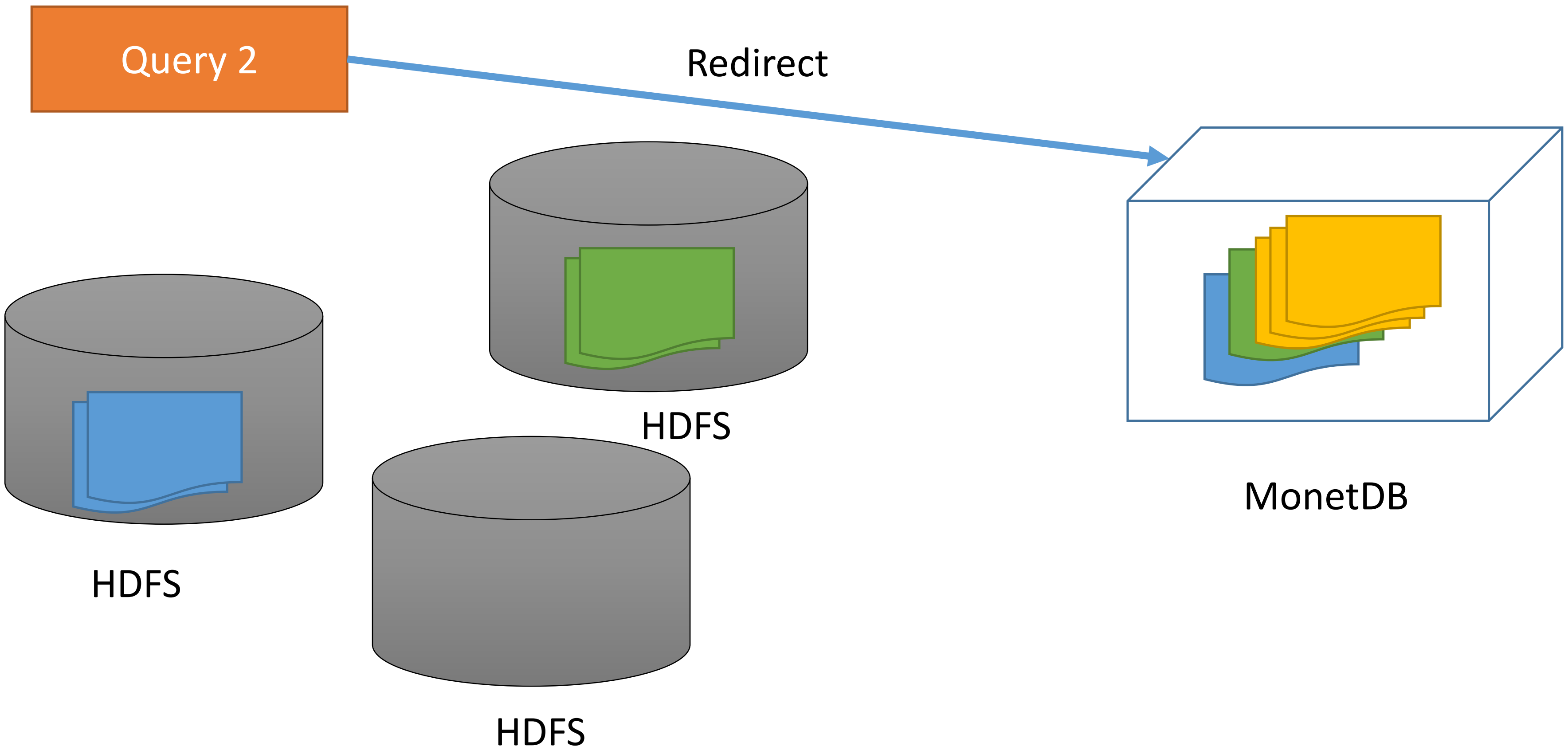
Work Flows



Work Flows



Work Flows



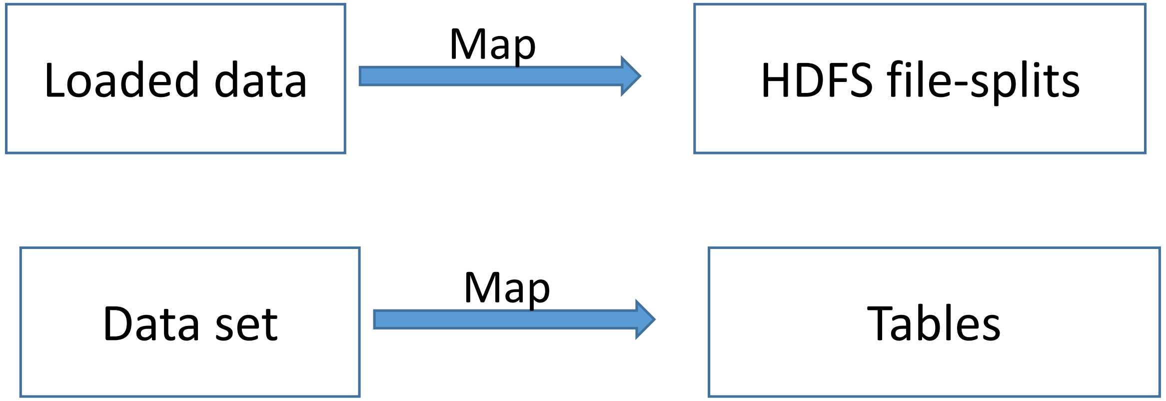
Invisible Loading

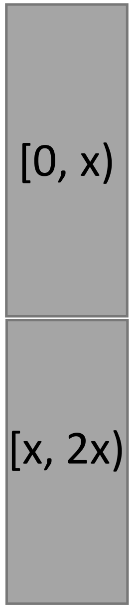
- Abstract, polymorphic Hadoop job (*InvisibleLoadJobBase*)
  - *Parser* object reads in input tuple to extract the attributes
  - Generate flexible schema

```
Table name: <file_name>_<parser_name>;
Schema: (1 <type>, 2 <type>, ...,
         n <type>);
```

Invisible Loading

- Catalog
  - Address Column enables alignment of partially loaded cols with other cols





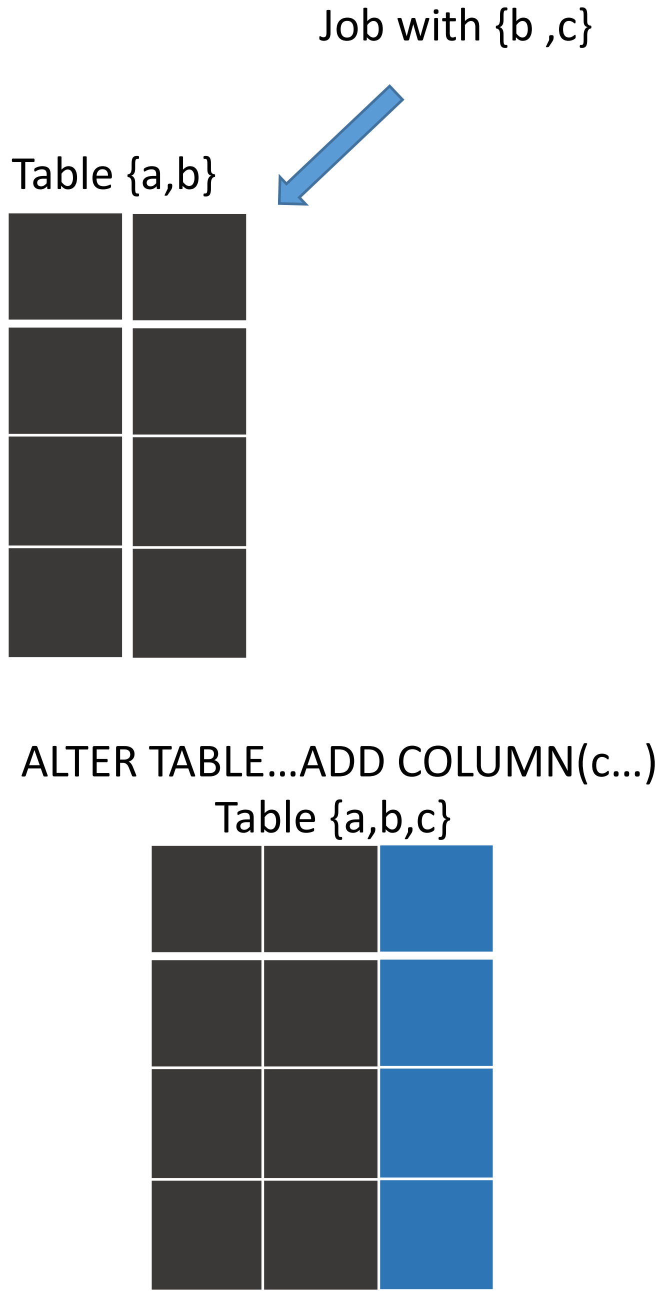
Address col

  - If table does not exist

    SQL CREATE TABLE

Incrementally Loading Attributes

- Loading attributes that are actually processed
  - *SQL ALTER TABLE…*
  - Size of Partition loaded per IL could be configured
  - Use Column store to avoid physically restructuring



Incremental Data Reorganization

- Pre-sorting is expensive and inflexible
  - Bad index results in poor query execution plans
  - All or nothing service
  - Take long time creating a complete index

Incremental Merge Sort

Based on basic **two-way external merge sort** algorithm

Basic two-way external features:

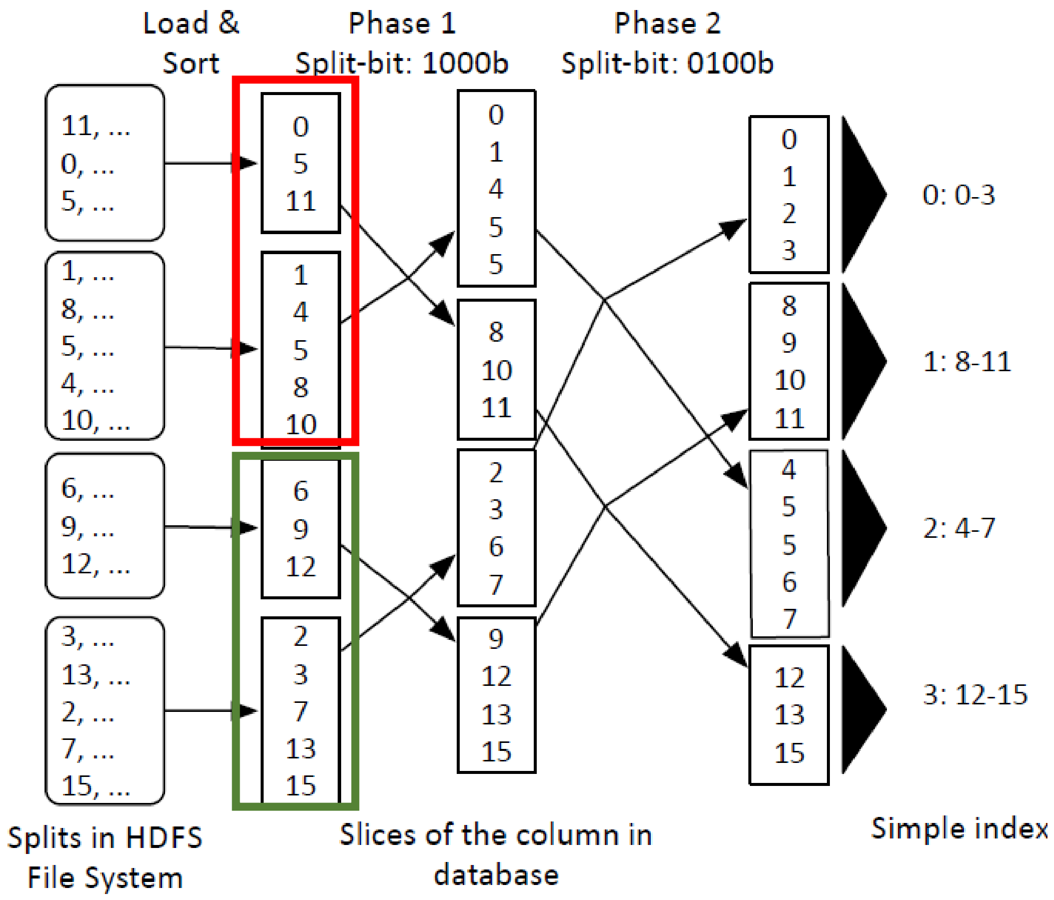
- Twice the amount of merge work than previous phase

- **Defeats** the key feature of any incremental strategy
  - Keep equal or less effort for any query in comparison to previous queries

Incremental Merge Sort

Goal: perform a bounded # of comparisons

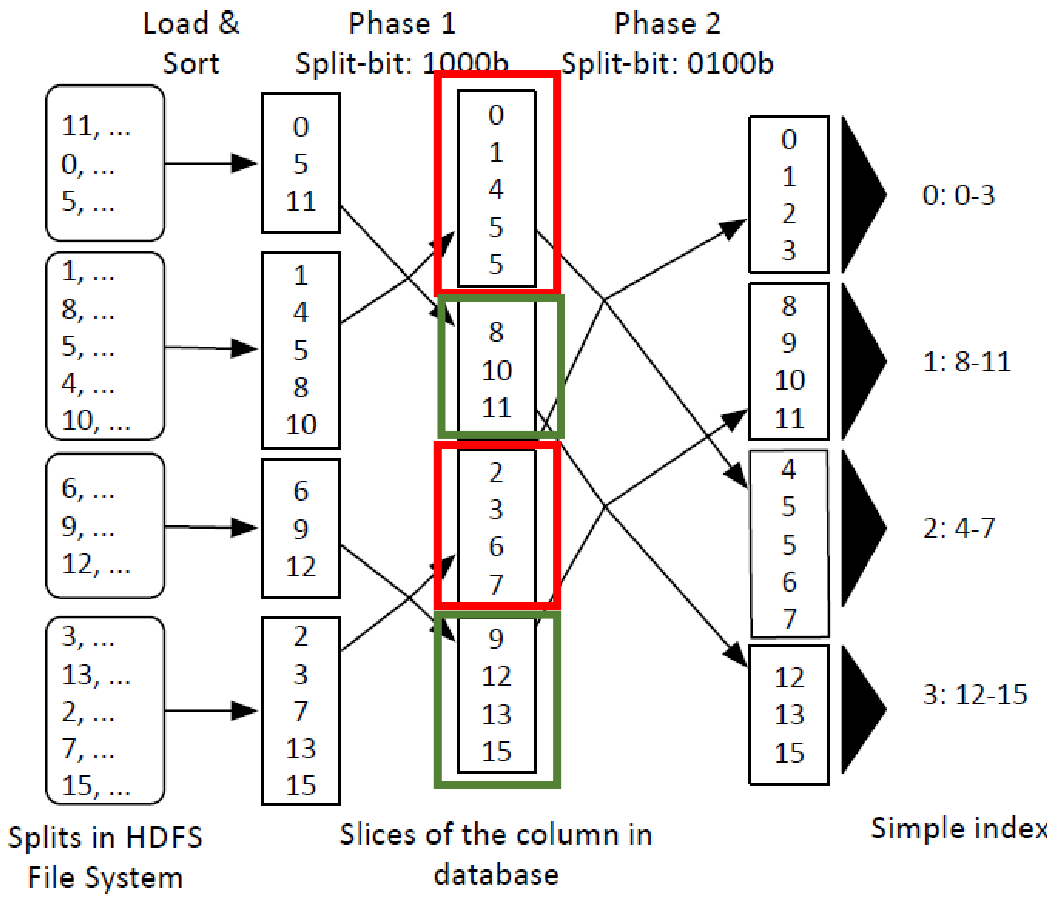
- Split-bit
- Go through *logk* phases of *k/2* merge/split operations on average *2*n/k* tuples
- Disjoint ranges



Incremental Merge Sort

Goal: perform a bounded # of comparisons

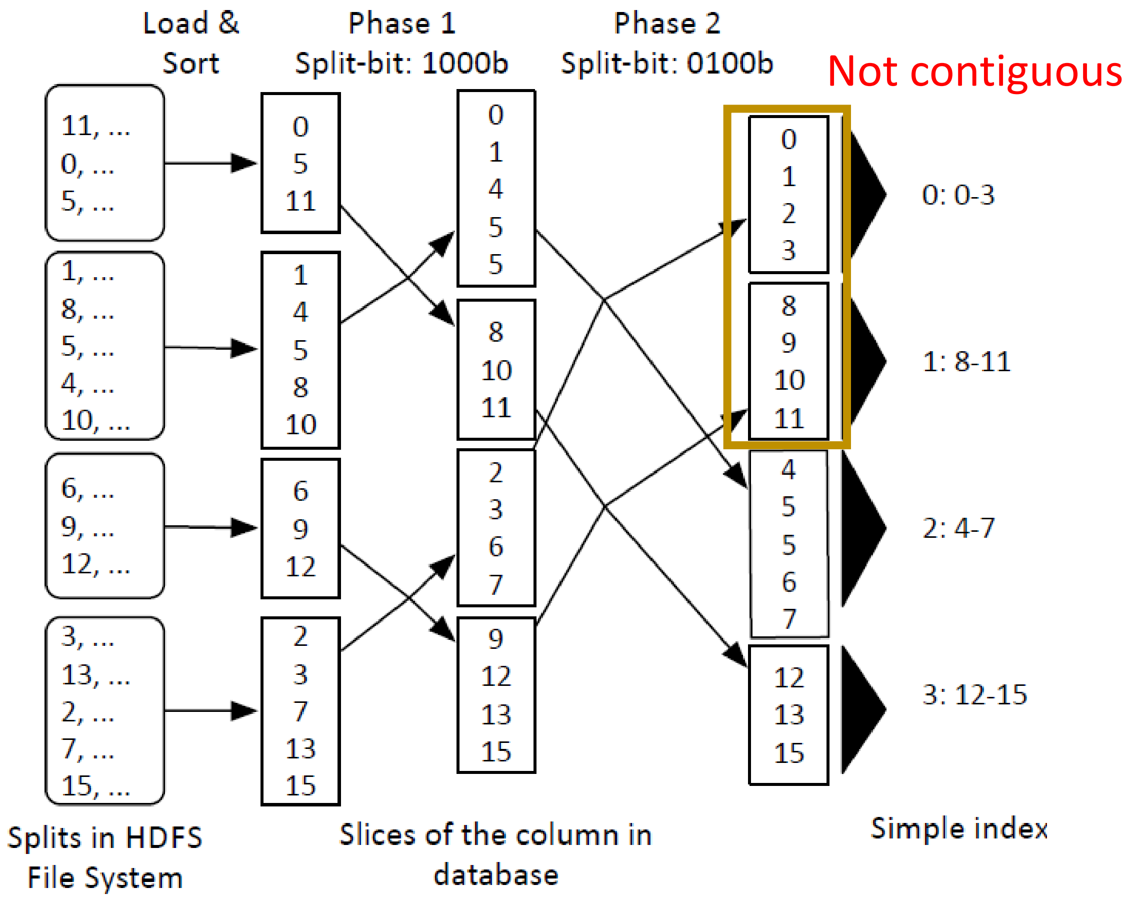
- Split-bit
- Go through *logk* phases of *k/2* merge/split operations on average *2*n/k* tuples
- Disjoint ranges



Incremental Merge Sort

Goal: perform a bounded # of comparisons

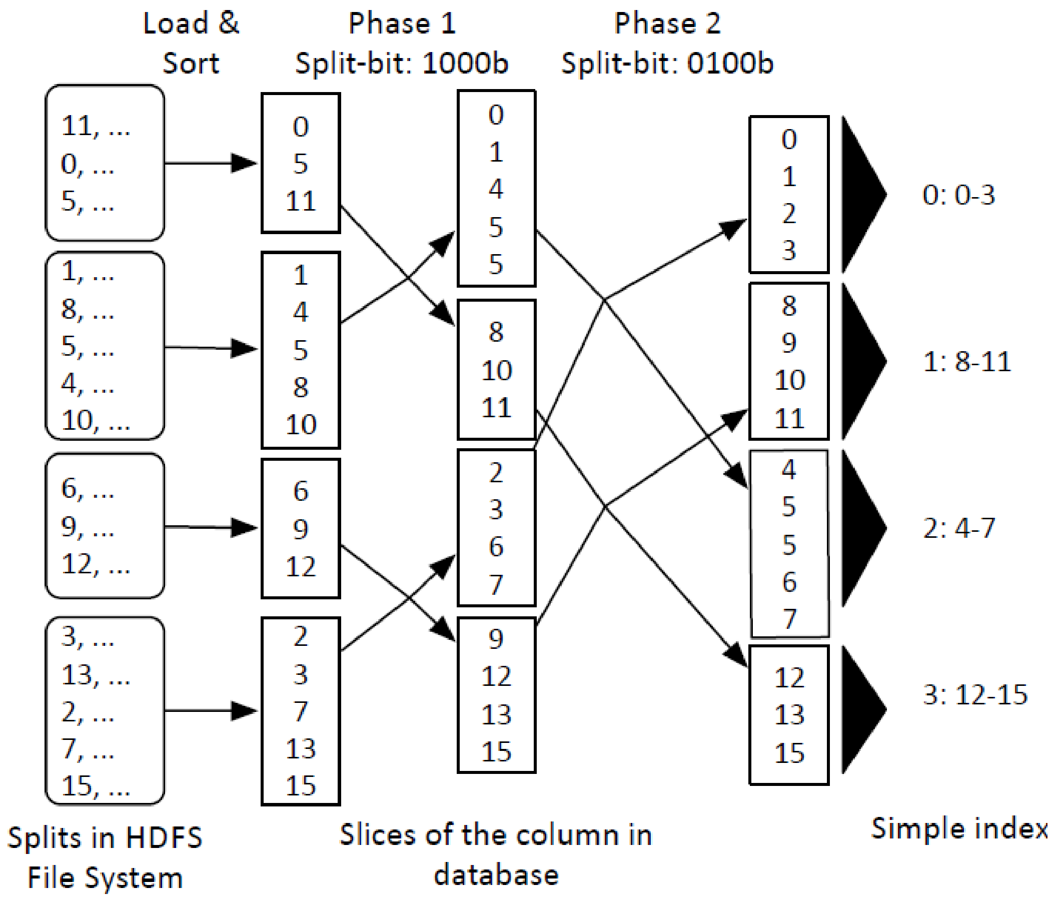
- Split-bit
- Go through *logk* phases of *k/2* merge/split operations on average *2*n/k* tuples
- Disjoint ranges



Incremental Merge Sort

Problem with this algorithm

- Create physical copy of columns with no GC
- Data skew
- Not query driven, all tuples are equally important



Integration Invisible Loading with Incremental Reorganization

- Frequency of access of a particular attribute determines how much it is loaded
  - Tuple-identifier(OIDs): determine how much of a column has been loaded

- Filtering operations on a particular attribute cause sort on the attribute’s column
  - Address Columns: track the movement of tuples due to sorting

Integration Invisible Loading with Incremental Reorganization

- Rules for reorganization at different loading states
  - Columns are completely loaded and sorted in the same order
    - Simple linear merge
  - Reconstruct a partially loaded columns with other columns.
    - Join on address column of primary column with OIDs of partially loaded columns
  - Sort a column to a different order
    - A copy for that column is created and use address column to track the movements

Integration Invisible Loading with Incremental Reorganization

X: {**a**, b}
Y: {**a**, c}
Z: {**b**, d}
At most one split is loaded per job per node

- Case 0: XXXX-YYYY
  - b is positionally aligned with a. no need OID
  - Tuple-identifier matching $\pi_{a,c}\left(\sigma_{f(a)}(a, addr_a) \bowtie (oid_c, c)\right)$
  - C drops OID after complete loading, and align with a

Integration Invisible Loading with Incremental Reorganization

X: {**a**, b}
Y: {**a**, c}
Z: {**b**, d}
At most one split is loaded per job per node

- Case 1: XX-YYYY-XX
  - b is positionally aligned with a
  - Tuple-identifier matching $\pi_{a,c}\left(\sigma_{f(a)}(a, addr_a) \bowtie (oid_c, c)\right)$
  - a is immediately sort
  - b create OID after third Y
  - c drops OID after fourth Y

Integration Invisible Loading with Incremental Reorganization

X: {**a**, b}
Y: {**a**, c}
Z: {**b**, d}
At most one split is loaded per job per node

- Case 2: {case 0 | case 1} - ZZZZ
  - A copy of b is created as b' $\pi_{b,d}\left(\sigma_{f(b)}(b, addr_a) \bowtie (oid_d, d)\right)$.
  - Addr{b} keeps track of b'

Integration Invisible Loading with Incremental Reorganization

X: {**a**, b}
Y: {**a**, c}
Z: {**b**, d}
At most one split is loaded per job per node

- Case 3: XX-ZZZZ-XX
  - Addr{a} for a and Addr{b} for b'
  - The following X load a from HDFS, and copy b within database to keep alignment with a

Experiments

Two extreme Example

- SQL Pre-load
- MapReduce

Two Dimensions:

- Vertically
- Horizontally

	Strategy	Description
1	SQL Pre-load	Pre-load the entire dataset into the database using SQL's 'COPY INTO' command. Data are sorted after loading using 'ORDER BY'.
2	Incremental Reorganize (all)	Load the entire dataset into the database system upon its first access, but unlike Pre-load above, do not immediately sort the data. Instead, data are incrementally reorganized as more queries access the data.
3	Incremental Reorganize (subset)	Same as Incremental Reorganize (all), except that only those attributes that are accessed by the current MapReduce job are loaded.
4	Invisible Loading (all)	The invisible loading algorithm described in Section 2, except that all attributes are loaded into the database (instead of the subset accessed by a particular MapReduce job).
5	Invisible Loading (subset)	The complete invisible loading algorithm described in Section 2.
6	MapReduce	Process the data entirely in Hadoop without database loading or reorganization. This is the performance the user can expect to achieve if data are never loaded into a database system

Table 1: Loading Strategies

Loading Experiments

Invisible Loading(2/5)
The response time is almost the same
With MR, but has a better improvement
In the next 10 jobs

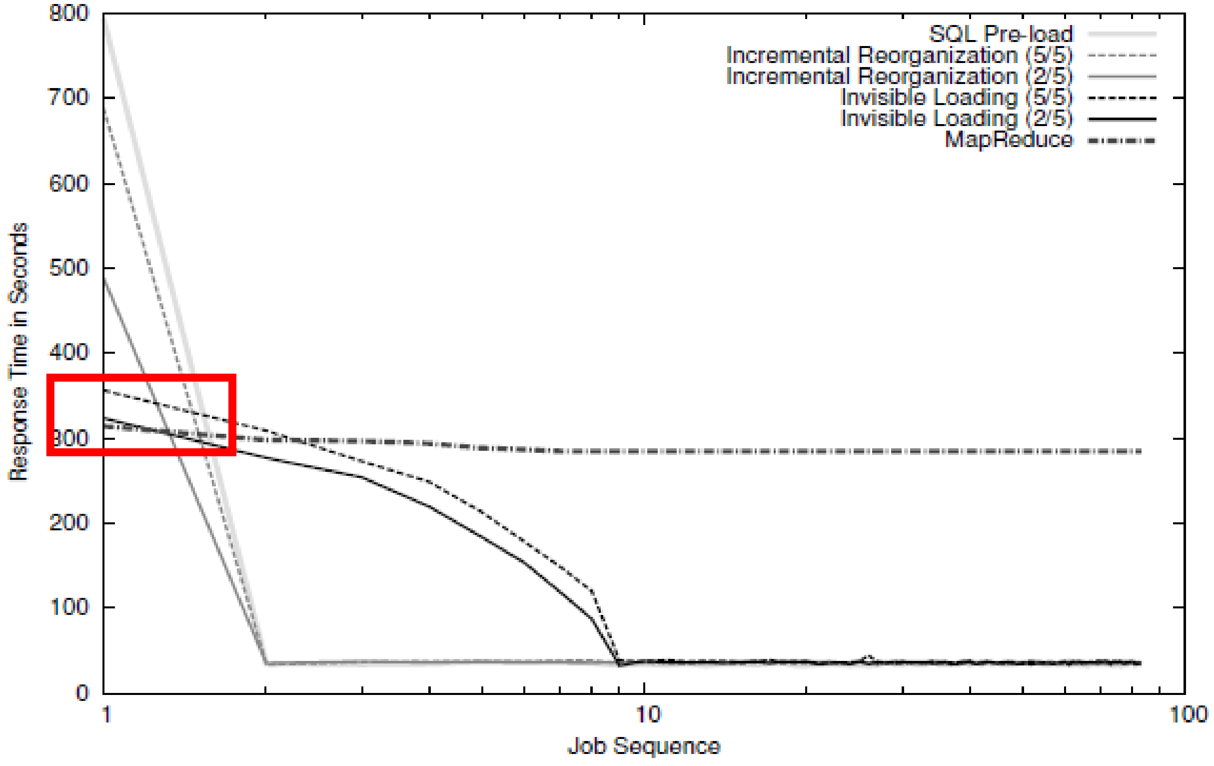


Figure 2: Response time of repeatedly executing selection queries over attributes a_0, a_1.

Loading Experiments

Invisible Loading:

- Low upfront cost of pre-loading
- Performs better when data are completely loade

Incremental reorganization

- Approximately the same with pre-load
  Sort in one go has little cumulative benefit

(2/5)Incremental reorganization

- Best cumulative effort if the other 3
  attributes are not accessed

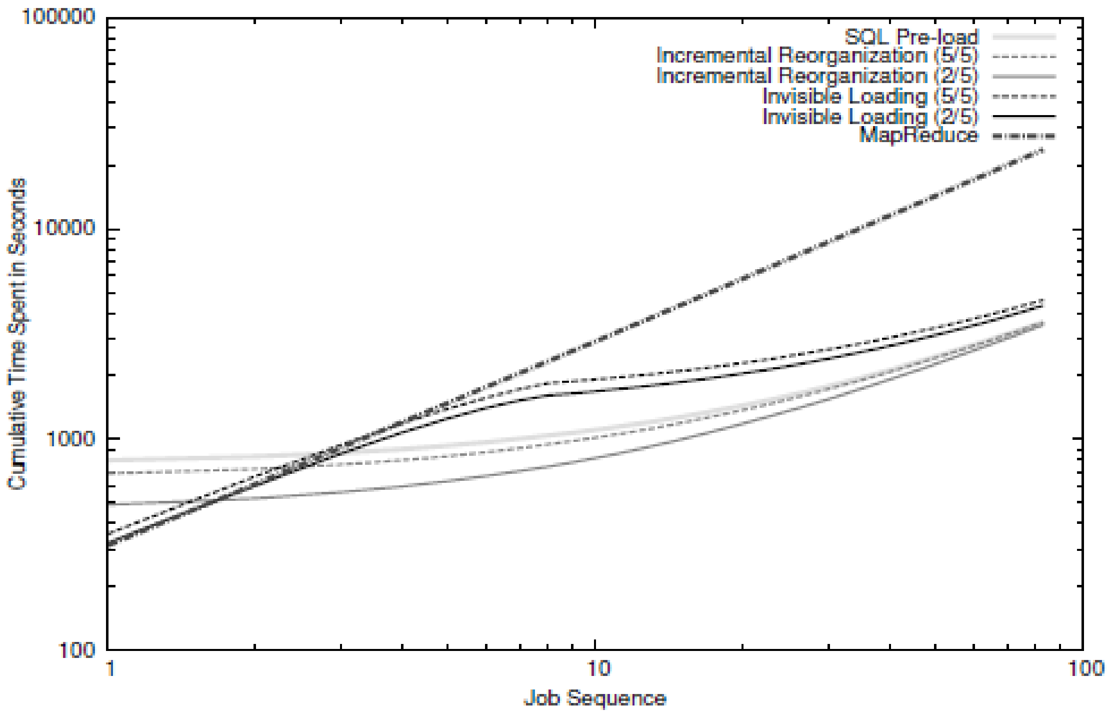


Figure 3: Cumulative cost of repeatedly executing selection queries over attributes a_0, a_1 (Experiment 1).

Summary

Strong Points:

- Almost no burden on MapReduce jobs
- Optimized data access for future analysis
- Relatively low cumulative cost in comparison to no data access

Weak Points:

- Data duplication cost, no GC
- Suitable for short-lived data

Thanks