Eagle-Eyed Elephant (E3): Split-Oriented Indexing in Hadoop

Mohamed Eltabakh
Worcester Polytechnic Institute, MA, USA

Joint work with IBM Almaden, CA, USA
F. Özcan, Y. Sismanis, H. Pirahesh, P. Haas, J. Vondrak

Data Explosion

1.3 Billion RFID tags in 2005
***30 Billion* RFID**
tags by 2010

***2 Billion* Internet**
users by 2011

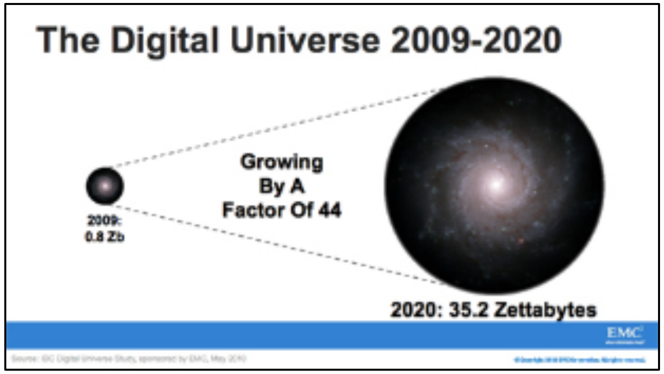
4.6 Billon
Mobile Phones
World Wide

Capital market
data volumes grew
***1,750%*, 2003-06**

Twitter process
***7 terabytes* of**
data every day

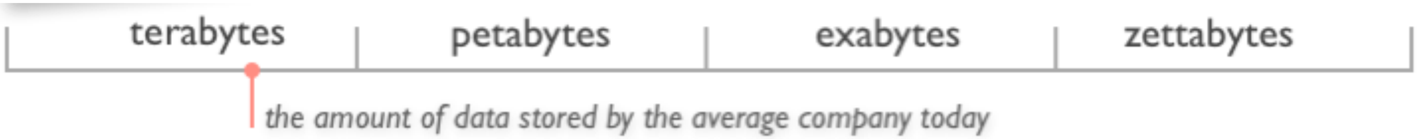
World Data Centre for Climate
- ***220 Terabytes* of Web data**
- ***9 Petabytes* of additional data**

Facebook process
***10 terabytes* of**
data every day



The Digital Universe 2009-2020

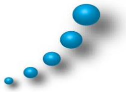
Source: IDC Digital Universe Study, sponsored by EMC, May 2010



the amount of data stored by the average company today

Hadoop Analytical Platform

- Hadoop is a software platform for *distributed processing* over:
  - ***Large datasets*** → Terabytes or petabytes of data
  - ***Large clusters*** → hundreds or thousands of nodes



Scalability (petabytes of data, thousands of machines)

Flexibility in accepting all data formats (no schema)

Efficient and simple fault-tolerant mechanism

Commodity inexpensive hardware

Hadoop: Poor Performance

- **Big performance gap between Hadoop and parallel databases**

E3 System addresses the 1st type of limitations (while retaining Hadoop's desired properties)

Many lessons from DBMSs are not utilized in Hadoop

>> Indexing, caching, materialization, partitioning, …

Expensive operations inherent to Hadoop's design

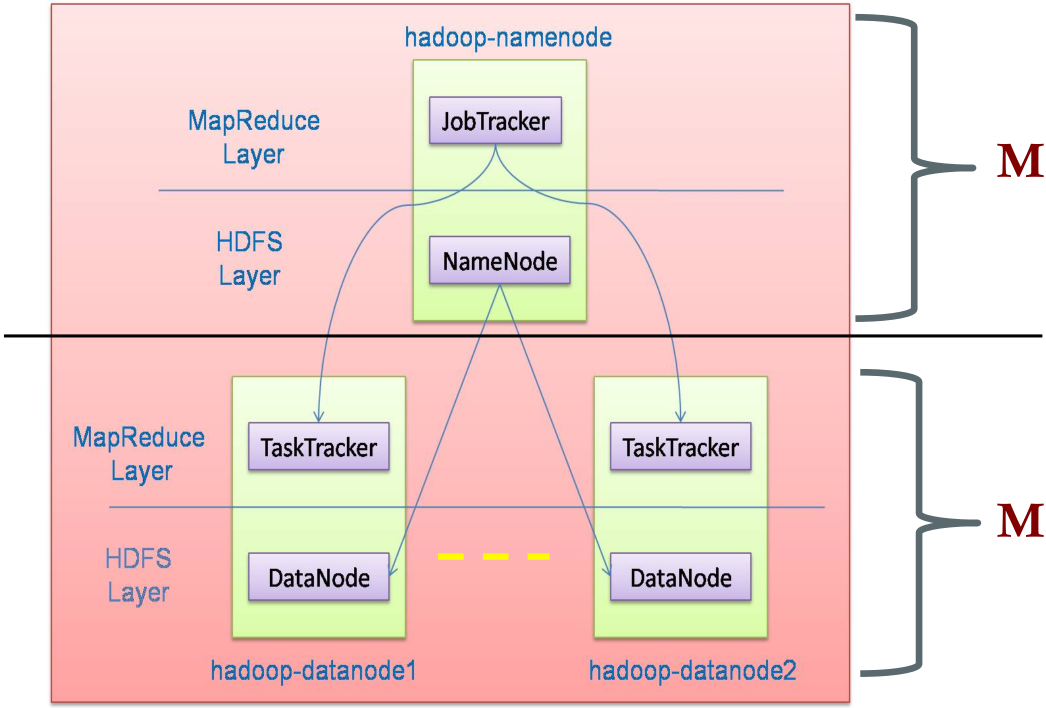
>> Blocking operators, disk-intensive use, no pipelining, …

Talk Outline

- **Background and Motivation**
- **E3 System Features**
  - **Indexing and Domain Segmentation**
  - **Materialized Views**
  - **Adaptive Caching**
- **Performance and Evaluation**

Overview on Hadoop

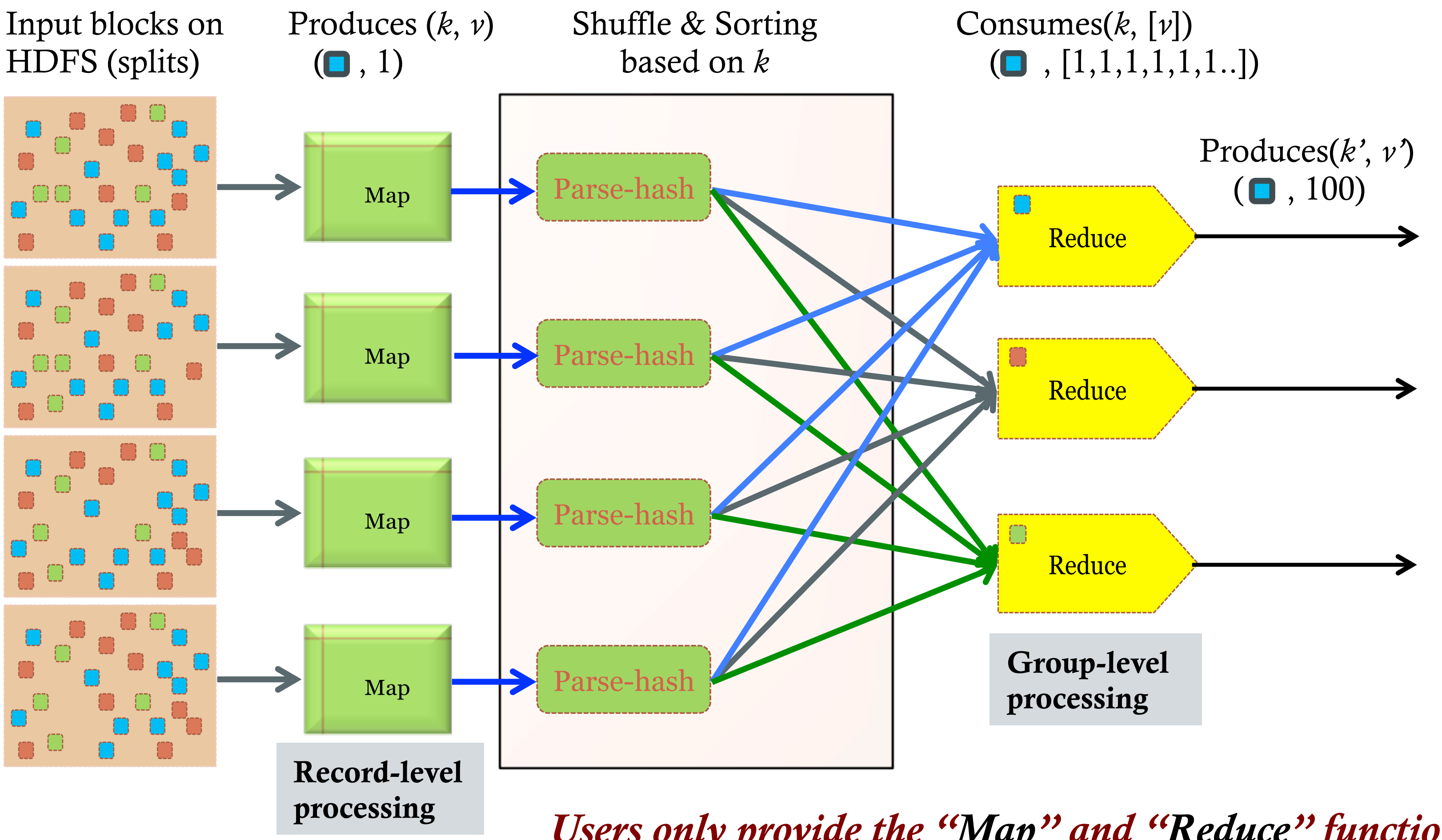
- **Hadoop is a *master-slave* *shared-nothing* *distributed* architecture**



Master node (single node)

Many slave nodes

Hadoop Execution Engine (Map-Reduce)



Input blocks on
HDFS (splits)

Produces (*k*, *v*)
(🟦, 1)

Shuffle & Sorting
based on *k*

Consumes(*k*, [*v*])
(🟦 , [1,1,1,1,1,1..])

Produces(*k'*, *v'*)
(🟦 , 100)

Map

Map

Map

Map

Parse-hash

Parse-hash

Parse-hash

Parse-hash

🟦 Reduce

🟥 Reduce

🟩 Reduce

Record-level processing

Group-level processing

Users only provide the "Map" and "Reduce" functions

E3 Motivation & Objectives

- **Typical Scenarios:** Analytical query workloads on Hadoop with *selection predicates*
  - Multiple (possibly repeated) queries over the same data set

- **No Smart Skipping:** No indexing (or *split elimination*) embedded into Hadoop
  - Queries scan all the data splits (relevant or not)

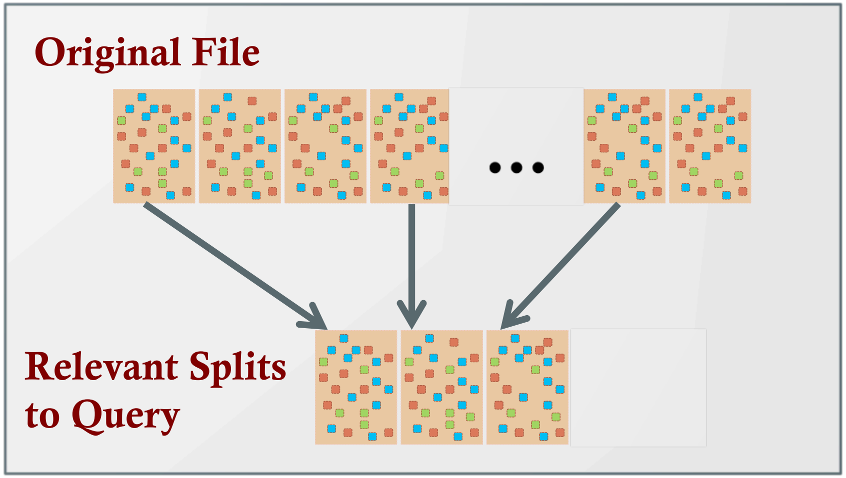
- **Little Users' Knowledge:** Workloads and data may change
  - Users may not know the query workload in advance or the data schema

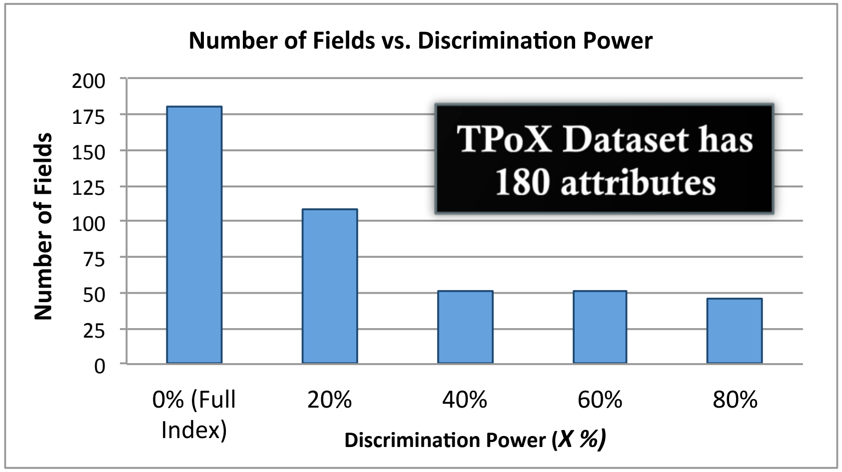
E3 Objectives

- **Discovery-based elimination of irrelevant splits**
- **No dependency on physical design, No data movement or DDL**
- **Adapt to workload and data changes**

E3 Design Goals

Re-think the indexing techniques and how they complement each other to fit Hadoop's environment





Split-Oriented Elimination (I/O)

- HDFS is block oriented
- Record-level elimination is not effective

Cover All Discriminating Attributes

- Most attributes are discriminating
- Go beyond the partitioning key(s)

Talk Outline

- **Background and Motivation**
- **E3 System Features**
  - **Indexing and Domain Segmentation**
  - **Materialized Views**
  - **Adaptive Caching**
- **Performance and Evaluation**

E3: Highlights

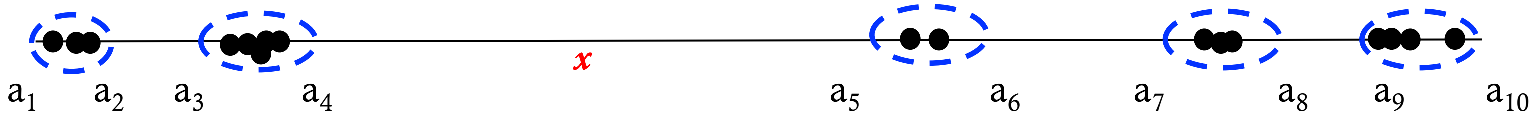
- **JSON-Based Data Model**
  - Works on all data types/sources that provide a mapping to JSON (JSON view of the data)
- **Pre-Processing Phase for each dataset**
  - Split-level statistics
  - Integration of several techniques

```
{
    "firstName": "John",
    "lastName": "Smith",
    "age": 25,
    "address": {
        "streetAddress": "21 2nd Street",
        "city": "New York",
        "state": "NY",
        "postalCode": "10021"
    },
    "phoneNumber": [
        {
            "type": "home",
            "number": "212 555-1234"
        },
        {
            "type": "fax",
            "number": "646 555-4567"
        }
    ]
}
```

- **Split elimination at I/O layer (InputFormat) before creating map tasks**
  - Can be integrated into Jaql
  - Can be used in hand-coded map-reduce jobs

1) Split-Level Domain Segmentation

- **Applied for all *numeric* and *date* attributes**
- **One-dimensional clustering to produce *multiple ranges* (Reduces false-negative hits)**
- **Given *k*, find the largest *k-1* gaps in the data**

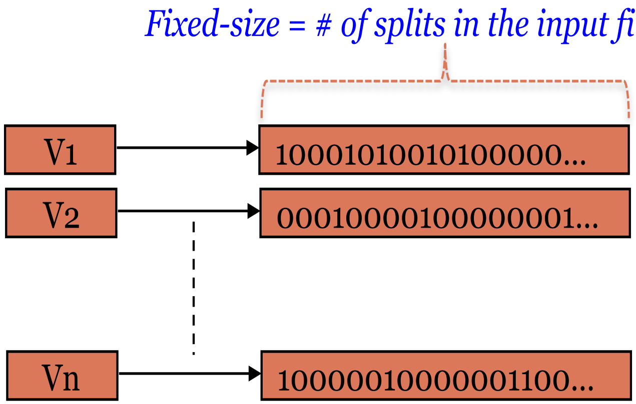


Query Q(x): $[a_1, a_{10}]$ contains x

$[a_1,a_2]$, $[a_3,a_4]$, $[a_5,a_6]$, $[a_7,a_8]$, $[a_9,a_{10}]$ do not contain x

2) Coarse-Grained Inverted Index

- **Split-level as opposed to record-level**
- **Inverted index implemented using bitmaps**
- **Run-Length Encoding for effective compression**

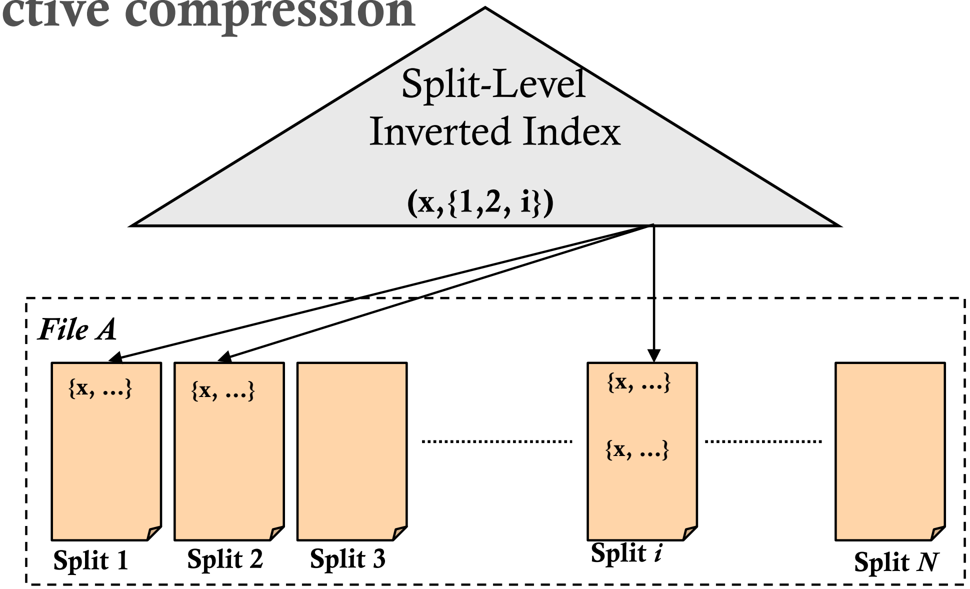


Fixed-size = # of splits in the input file

V1 → 10001010010100000...

V2 → 00010000100000001...

Vn → 10000010000001100...



Split-Level Inverted Index

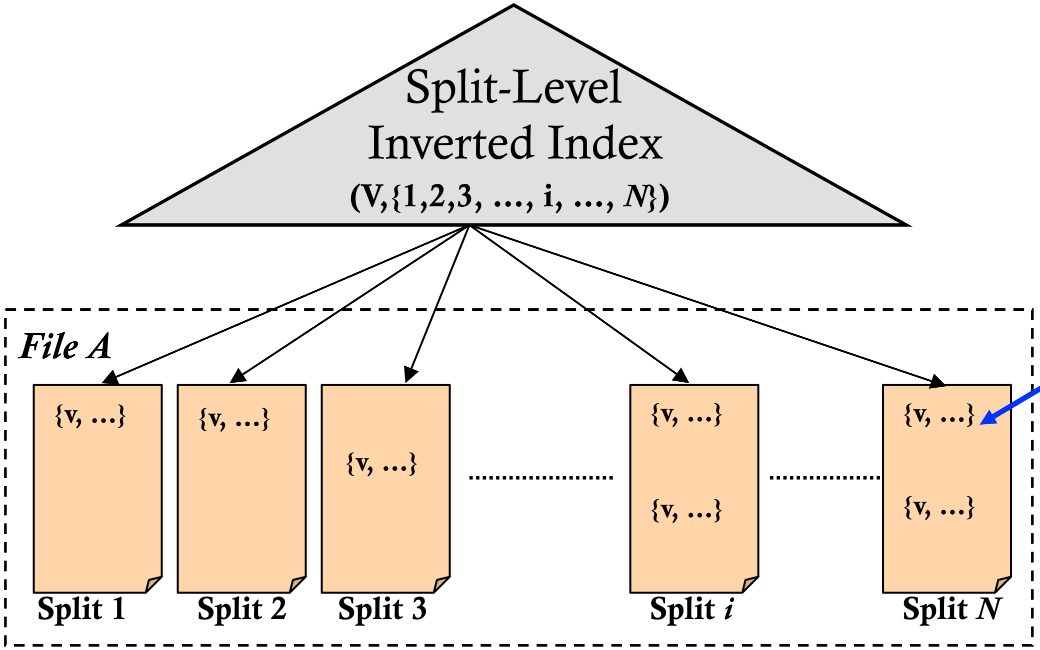
(x,{1,2, i})

File A

Query Q(*x*): Only read splits 1, 2, and *i*

Inverted Index Limitations

- **Inverted Index is of no use for *infrequent-scattered* values**
  - **Values appearing in *many splits*, but *few times* per split**

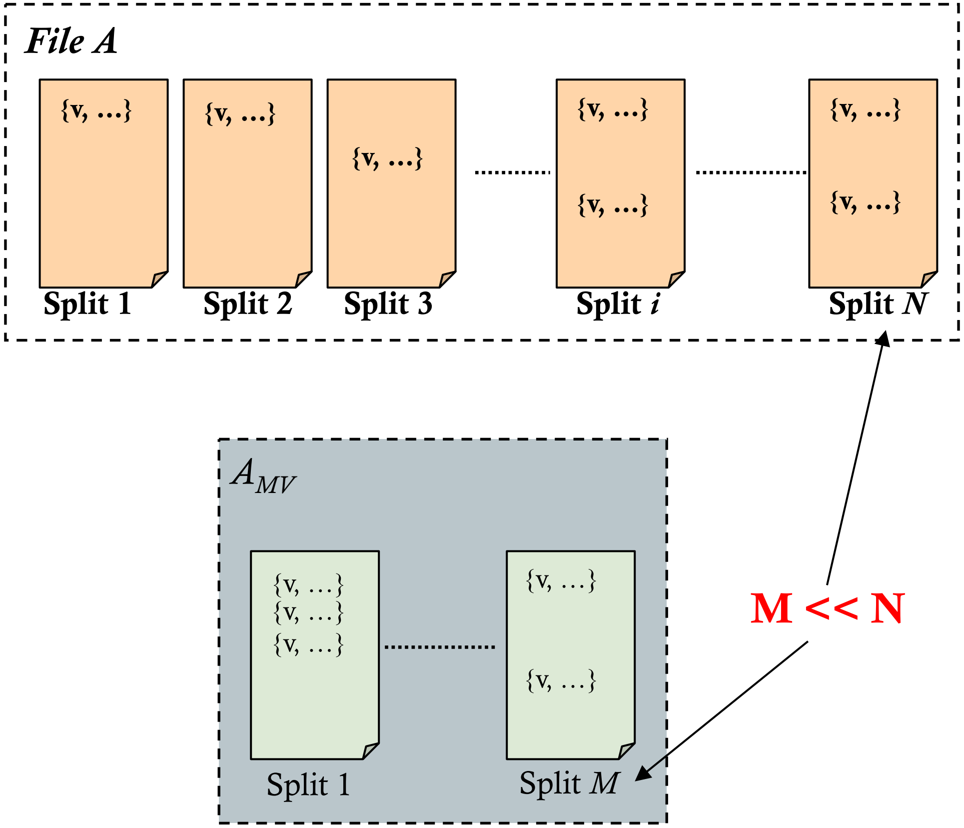


***v* is infrequent-scattered value**

Query Q(*v*): Must read all splits containing value *v* !

3) Materialized Views

- **Build a materialized view A_{MV} for each file A**
- **Copy the data records containing v to A_{MV}**
- **$|A_{MV}| \ll |A|$ (in splits)**
- **At query time, E3 re-directs Q(v) from A to A_{MV}**

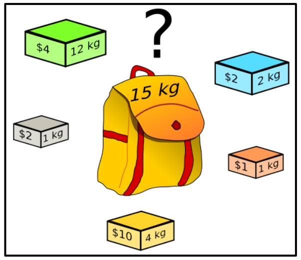


Query Q(v): read only M splits ($M \ll N$)

Building the Materialized View

- **MV is relatively very small** ➔ $|A_{MV}| \approx (1\%\text{-}2\%)\ |A|$
- **Infrequent-scattered values can be too many** ➔ *which v's to select?*

- **Modeling as optimization problem:** ***Submodular 0-1 Knapsack problem***
  - Space constraint: A_{MV} can hold M splits (R records)
  - Each value *v* has a ***profit*** and a ***cost***
    - |Splits(*v*)|: # splits containing value *v in original file A*
    - |Records(*v*)|: # records containing value *v* in *A*
    - ***Profit(v) = |Splits(v)| – M***
    - ***Cost(v) = |Records(v)|***



Select subset of values *v* to:
Maximize Σ profit(*v*) | Σ cost(*v*) <= R

Building the Materialized View: More Challenges

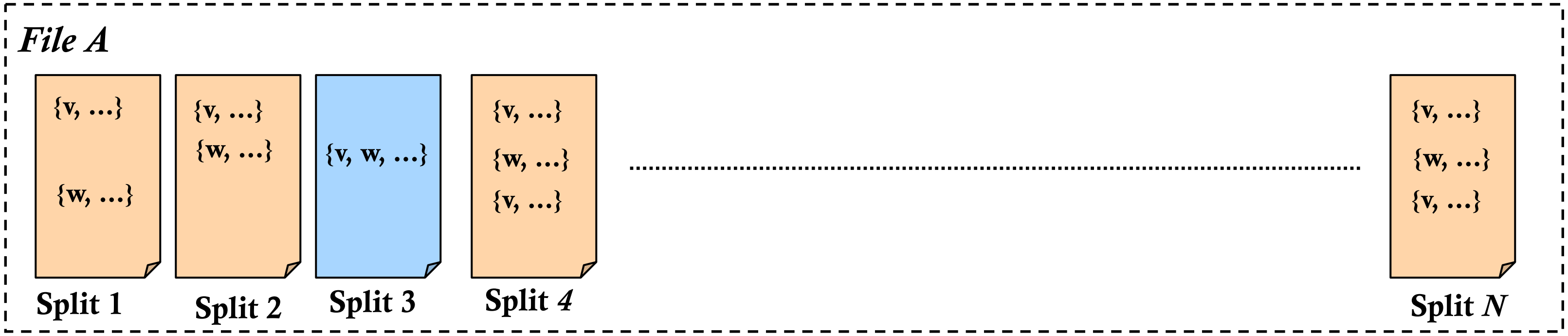
- **Submodular 0-1 Knapsack problem because**
  - Selecting v and copying its records to A_{MV} changes the cost of all other values v' contained in v's records
- **Naïve greedy algorithm is too expensive in Hadoop**
  - Requires sorting all the values (*w.r.t. profit/cost*) before selection
- **E3 avoids sorting**
  - Estimates an upper bound K values needed to fill in A_{MV} (over estimate)
  - One scan over the values ➔ maintain the ***top K in max-heap*** (*profit/cost*)
  - Select from the top K (in order) until A_{MV} is full

Talk Outline

- Background and Motivation
- E3 System Features
  - Indexing and Domain Segmentation
  - Materialized Views
  - Adaptive Caching
- Performance and Evaluation

Optimizing Conjunctive Predicates

- **Conjunctive predicates can be *together* very selective**
  - But also harder to optimize (each predicate by itself may not be selective)



File A

{v, …}
{w, …}
Split 1

{v, …}
{w, …}
Split 2

{v, w, …}
Split 3

{v, …}
{w, …}
{v, …}
Split *4*

{v, …}
{w, …}
{v, …}
Split *N*

Query Q(*v,w*) ➔ *read split 3 only*

- **Index cannot help:** splits(v) ∩ splits(w) = {1, 2, 3, …, N}
- **Materialized Views cannot help:** domain is too large to enumerate

Handling "nasty" Value-Pairs

- **Too expensive to identify all such value pairs (*v*, *w*)**
  - Require computing $|splits(v) \cap splits(w)| >> |splits(v,w)|$ for all (v,w) value pairs
- **Sampling does not work**
- **E3's Solution: Adaptive cache**
  - Only "cache" pairs that are:
    - Very nasty (high savings in splits if cached)
    - Referenced frequently
    - Referenced recently

4) Adaptive Caching for "nasty" Value-Pairs

- Select the value-pairs based on the ***observed query workload***
- **Given (Q = P1 and P2) over values *v* and *w***
  - Compute ($\text{splits}(v) \cap \text{splits}(w)$) from the inverted index
  - Monitor which map tasks return output records → $\text{splits}(v, w)$
  - If $|\text{splits}(v) \cap \text{splits}(w)| >> |\text{splits}(v, w)|$, then
    - Add (v, w, $\text{splits}(v, w)$) to the cache

Cache is limited in space, value-pairs can be too many

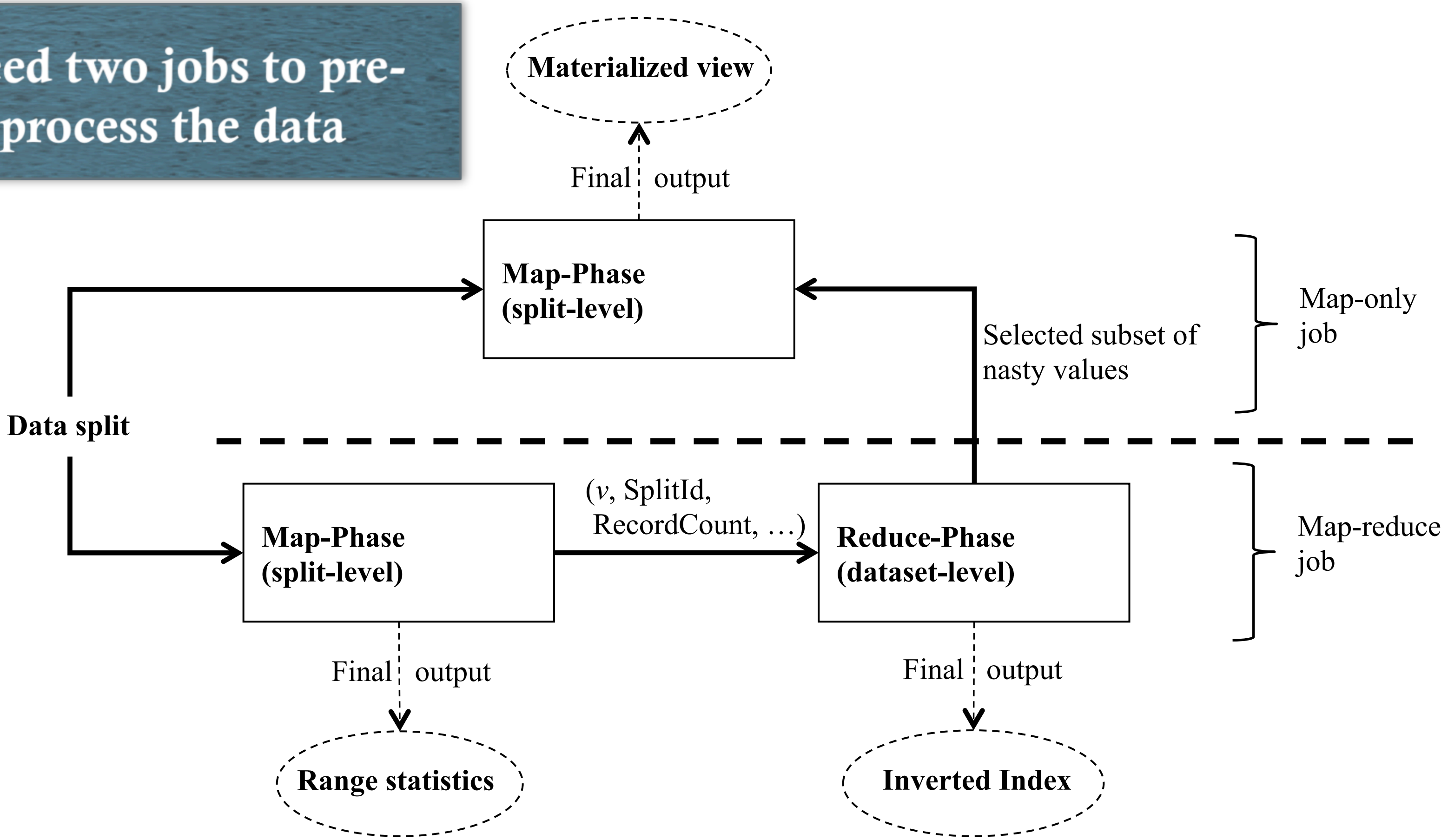
E3's Cache Replacement Policy

- **LRU may perform poorly**
  - It does not take savings into account

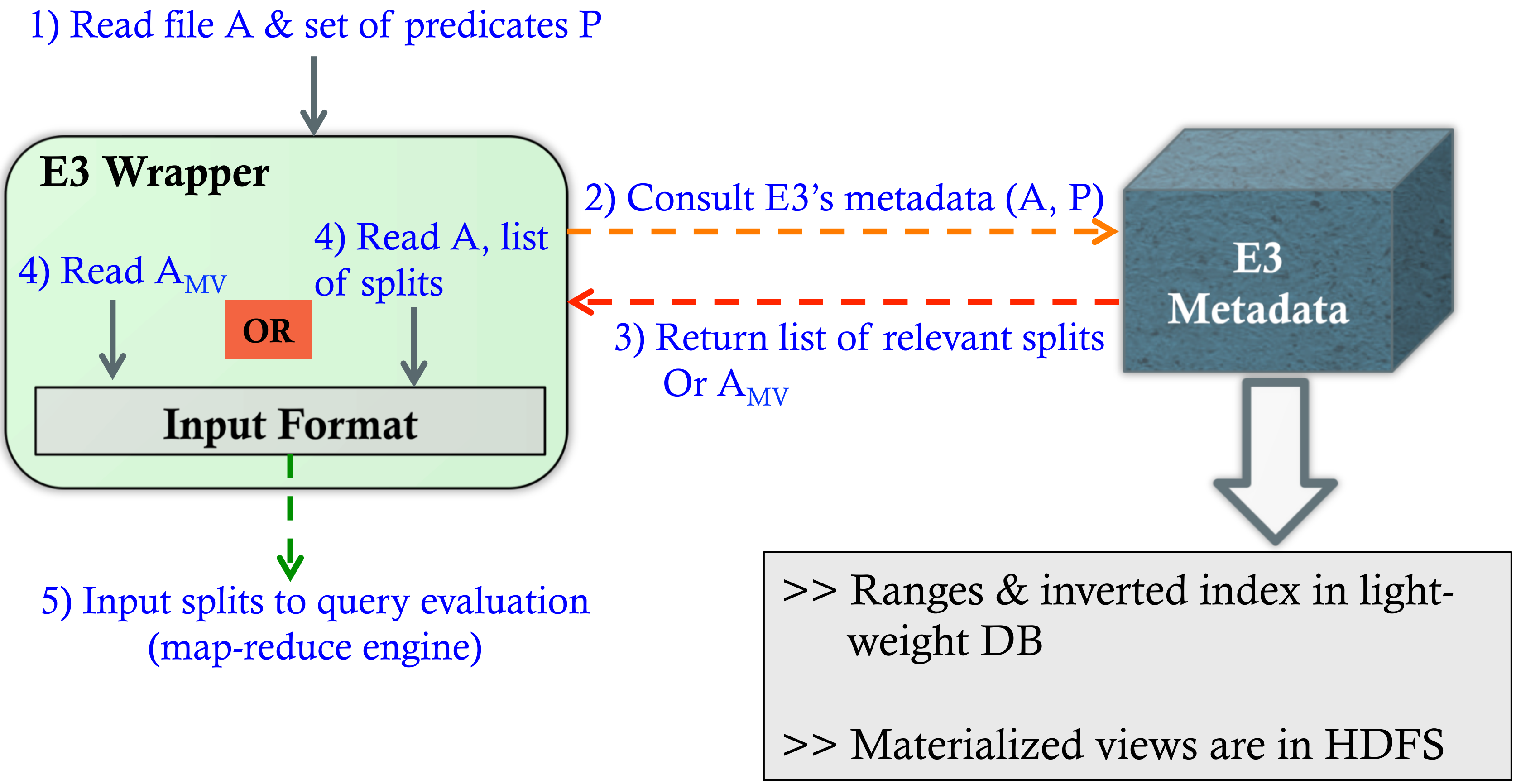
- **SFR (Savings-Frequency-Recency) Replacement Policy**
  - Compute a weight for candidate (v, w):
    - ***Savings in splits***: the bigger the saving, the higher the weight
    - ***Frequency:*** the more frequently queried, the higher the weight
    - ***Recency:*** the more recently queried, the higher the weight

E3 Computation Flow

Need two jobs to pre-process the data



E3 Query Evaluation
(Putting It All Together)



1) Read file A & set of predicates P

E3 Wrapper

4) Read A_{MV}

OR

4) Read A, list of splits

Input Format

2) Consult E3's metadata (A, P)

3) Return list of relevant splits Or A_{MV}

5) Input splits to query evaluation (map-reduce engine)

>> Ranges & inverted index in light-weight DB

>> Materialized views are in HDFS

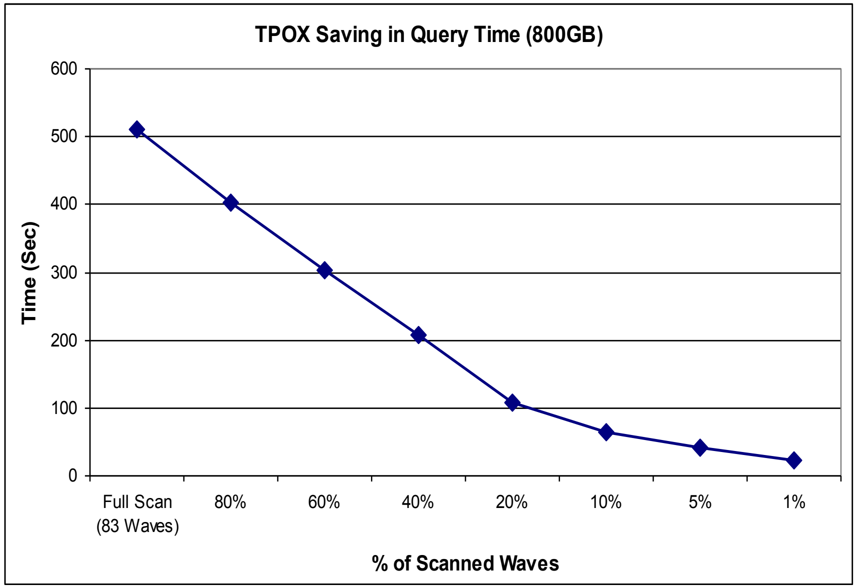
Talk Outline

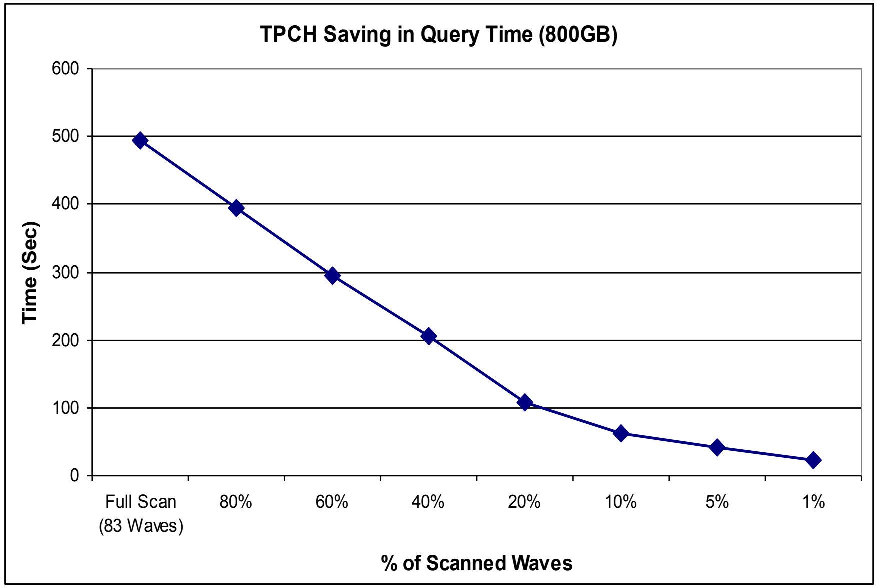
- **Background and Motivation**
- **E3 System Features**
  - **Indexing and Domain Segmentation**
  - **Materialized Views**
  - **Adaptive Caching**
- **Performance and Evaluation**

Experimental Setup

- **Datasets (800GB)**
  - Transaction Processing over XML (TPoX) – Orders
    - 4 levels of nesting, 181 distinct fields
  - Transaction Processing Council (TPCH) – LineItems
    - 1 level (no nesting),16 distinct fields
- **Cluster**
  - 41 nodes cluster: 1 master, and 40 data nodes, 8 cores
  - 160 Mappers and 160 Reducers
  - Block size = 64MB, Replication factor = 2
- **Performance**
  - Wall clock savings at query time
  - Computation cost of (1) Ranges, (2) Indexes, (3) Materialized view
  - Storage overhead of (1) Ranges, (2) Indexes, (3) Materialized view

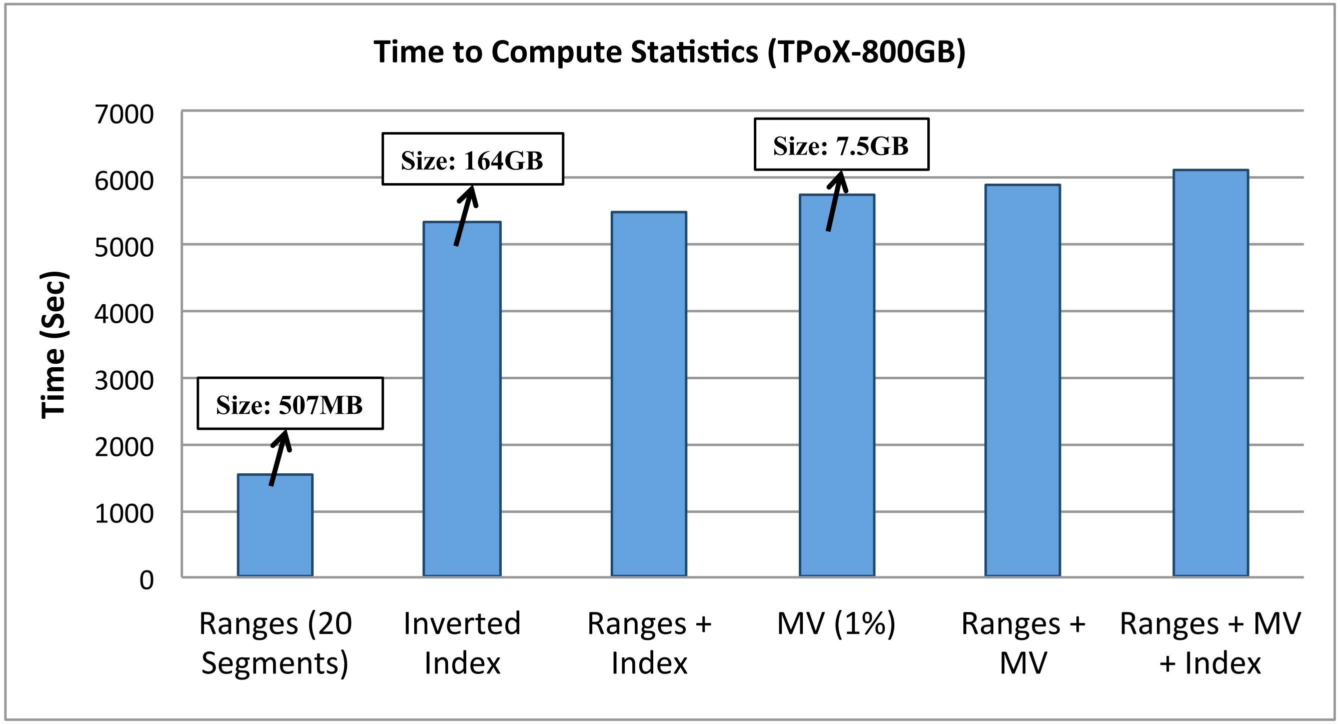
Query Response Time Savings





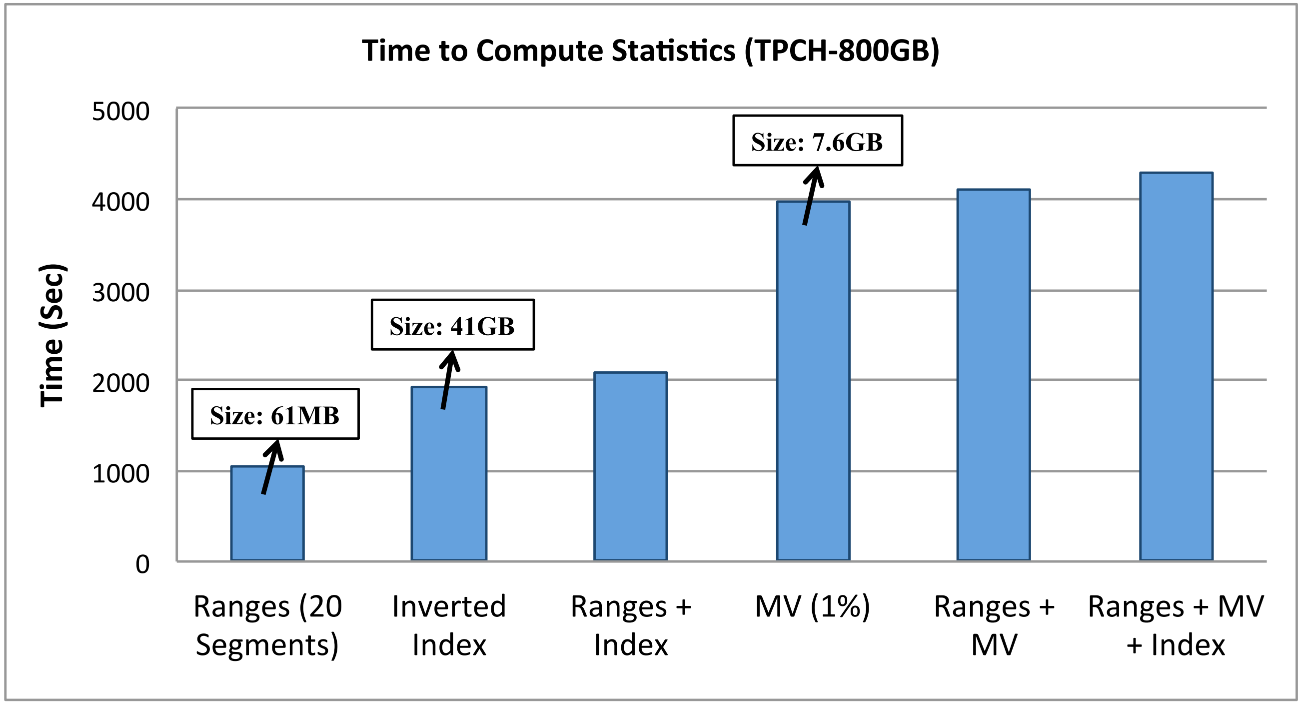
- **Query:** read(hdfs('input')) → filter (P1 ^ P2) → count();
  - Equality predicates
- Savings depend on selectivity → up to **20x with E3 optimizations**

Computation Cost (TPoX)



- **Costs are shared whenever possible**
- **Requires ~12 selective queries to redeem the cost**

Computation Cost (TPCH)



- **Requires ~8 selective queries to redeem the cost**

Summary & Lessons Learned

- **Eagle-Eyed Elephant (E3) *integrates various indexing* and elimination techniques to effectively eliminate splits (I/O)**
- **Up to *20x savings* can be achieved using E3 optimizations**
- **Discovery-based, No DDL or data movement**
- **Partitioning alone is not enough. Also indexing alone is not enough**
- **More complex data ➔ More preprocessing cost ➔ more queries to redeem the cost**

Related Work: Key Differences

- **Integration between multiple split-elimination techniques**
  - **Others use one mechanism**
- **Use of caching and materialized views is novel in Hadoop's environment**
- **Elimination of splits before reading them (I/O)**
  - **Others skip splits after retrieving them from disk**

Eagle-Eyed Elephant (E3): Split-Oriented Indexing in Hadoop

Thank You

?