UNIVERSITY OF WATERLOO

Big Data Infrastructure

CS 489/698 Big Data Infrastructure (Winter 2016)

Week 7: Analyzing Relational Data (3/3)

February 25, 2016

Jimmy Lin
David R. Cheriton School of Computer Science
University of Waterloo

These slides are available at http://lintool.github.io/bigdata-2016w/

MapReduce: A Major Step Backwards?

- MapReduce is a step backward in database access:
  - Schemas are good
  - Separation of the schema from the application is good
  - High-level access languages are good
- MapReduce is poor implementation
  - Brute force and only brute force (no indexes, for example)
- MapReduce is not novel
- MapReduce is missing features
  - Bulk loader, indexing, updates, transactions…
- MapReduce is incompatible with DMBS tools

Source: Blog post by DeWitt and Stonebraker

Hadoop vs. Databases: Grep

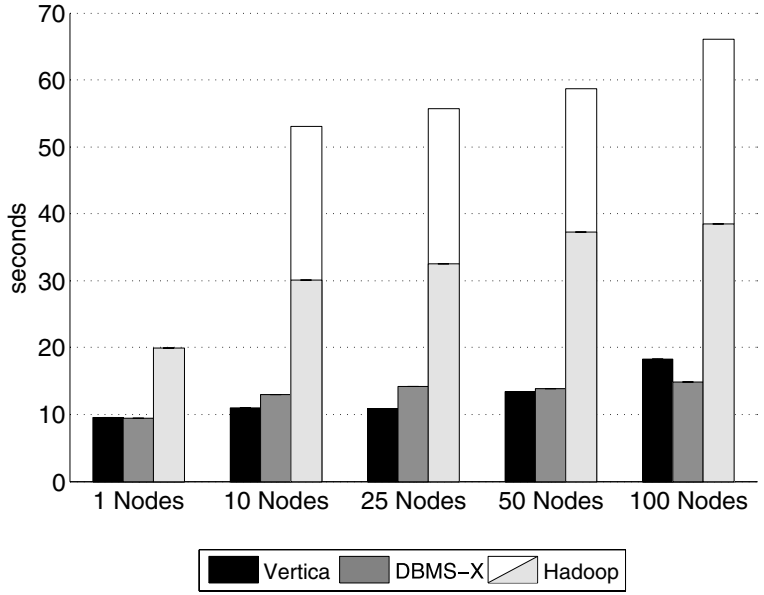


Figure 4: Grep Task Results – 535MB/node Data Set

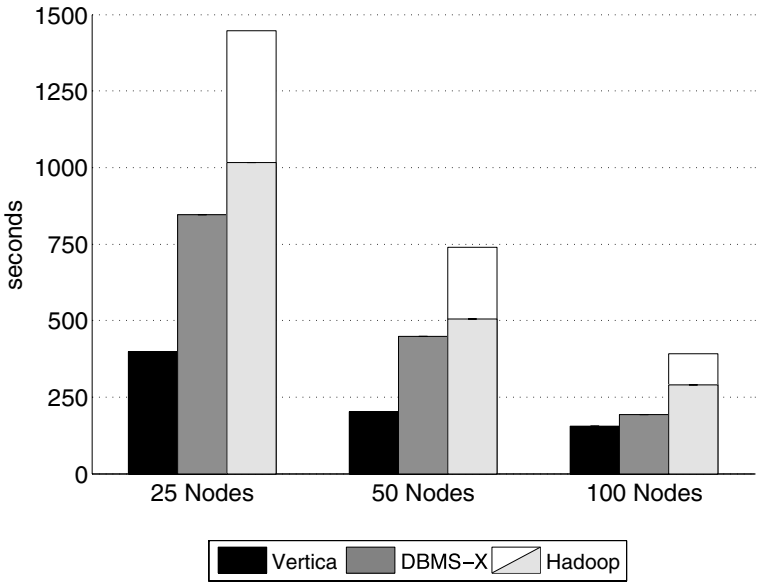


Figure 5: Grep Task Results – 1TB/cluster Data Set

```
SELECT * FROM Data WHERE field LIKE '%XYZ%';
```

Source: Pavlo et al. (2009) A Comparison of Approaches to Large-Scale Data Analysis. SIGMOD.

Hadoop vs. Databases: Select

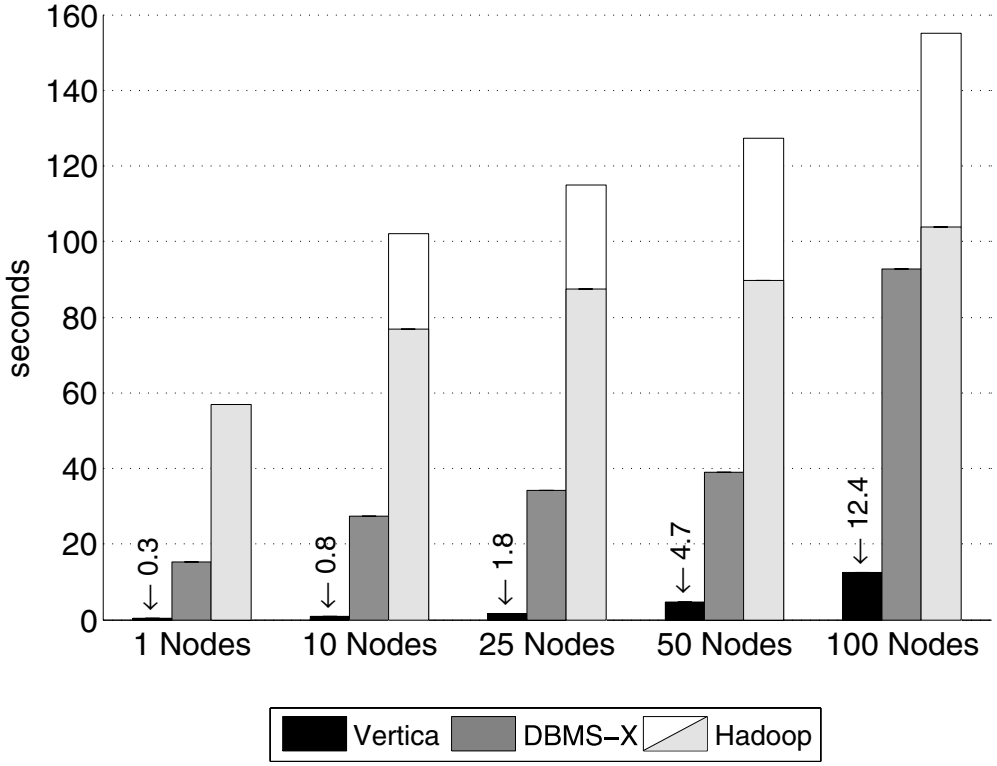


Figure 6: Selection Task Results

```
SELECT pageURL, pageRank
FROM Rankings WHERE pageRank > X;
```

Source: Pavlo et al. (2009) A Comparison of Approaches to Large-Scale Data Analysis. SIGMOD.

Hadoop vs. Databases: Aggregation

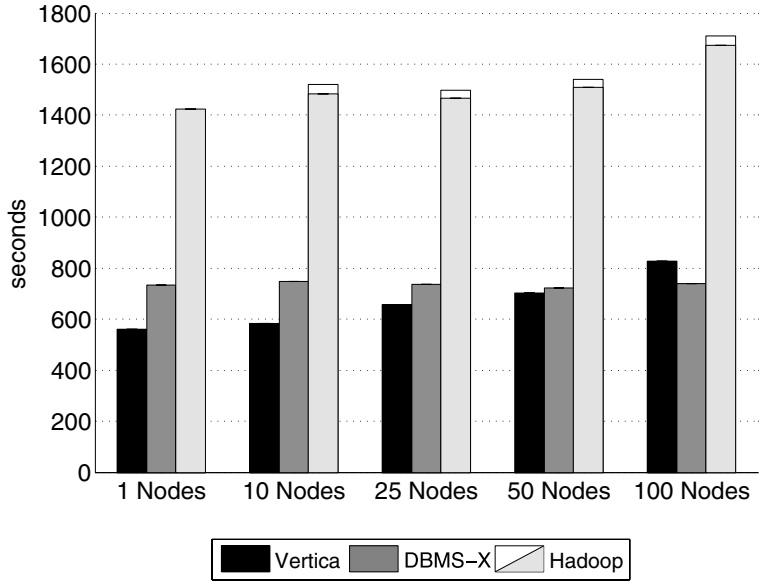


Figure 7: Aggregation Task Results (2.5 million Groups)

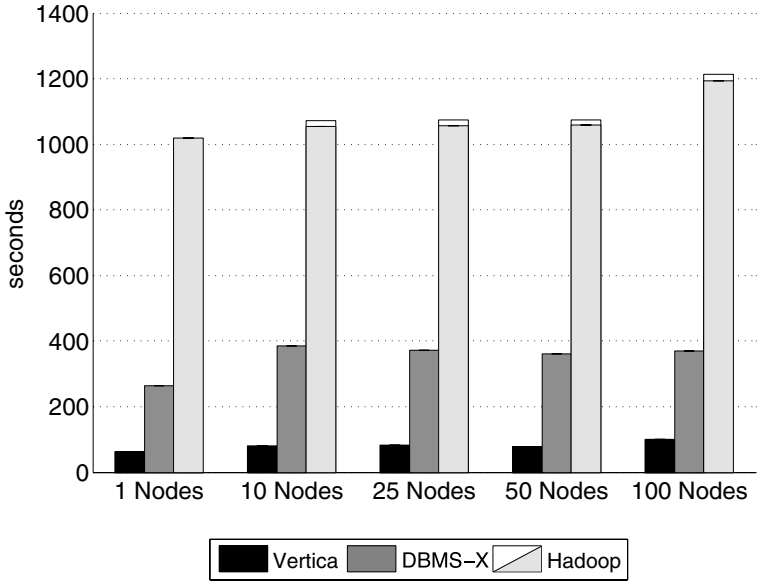


Figure 8: Aggregation Task Results (2,000 Groups)

```
SELECT sourceIP, SUM(adRevenue)
FROM UserVisits GROUP BY sourceIP;
```

Source: Pavlo et al. (2009) A Comparison of Approaches to Large-Scale Data Analysis. SIGMOD.

Hadoop vs. Databases: Join

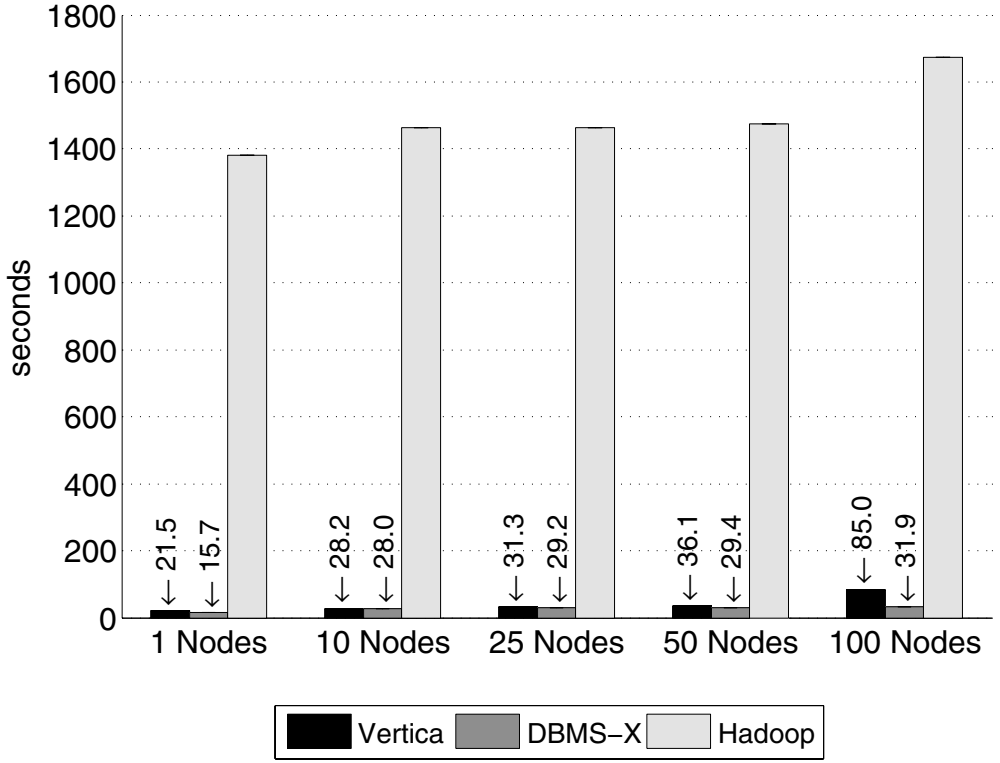


Figure 9: Join Task Results

```
SELECT INTO Temp sourceIP, AVG(pageRank) as avgPageRank, SUM(adRevenue) as totalRevenue
FROM Rankings AS R, UserVisits AS UV
WHERE R.pageURL = UV.destURL AND UV.visitDate BETWEEN Date('2000-01-15') AND Date('2000-01-22')
GROUP BY UV.sourceIP;

SELECT sourceIP, totalRevenue, avgPageRank FROM Temp ORDER BY totalRevenue DESC LIMIT 1;
```

Source: Pavlo et al. (2009) A Comparison of Approaches to Large-Scale Data Analysis. SIGMOD.

Hadoop is slow...

Source: Wikipedia (Tortoise)

Source: Wikipedia (Fish)

Why was Hadoop slow?

`Integer.parseInt`

`String.substring`

`String.split`

Hadoop slow because string manipulation is slow?

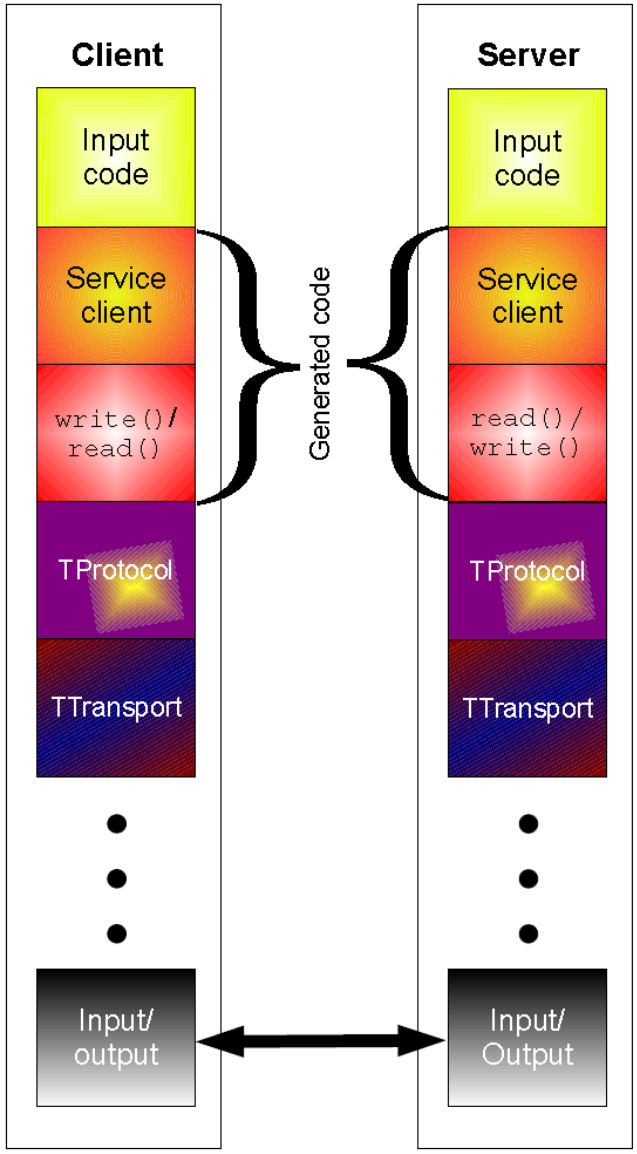
Key Ideas

- Binary representations are good
- Binary representations need schemas
- Schemas allow logical/physical separation
- Logical/physical separation allows you to do cool things

Thrift

- Originally developed by Facebook, now an Apache project
- Provides a DDL with numerous language bindings
  - Compact binary encoding of typed structs
  - Fields can be marked as optional or required
  - Compiler automatically generates code for manipulating messages
- Provides RPC mechanisms for service definitions
- Alternatives include protobufs and Avro

Thrift



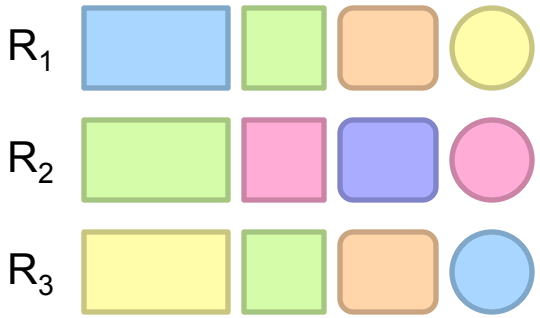

```
struct Tweet {
 1: required i32 userId;
 2: required string userName;
 3: required string text;
 4: optional Location loc;
}

struct Location {
 1: required double latitude;
 2: required double longitude;
}
```

Why not...

- XML or JSON?
- REST?

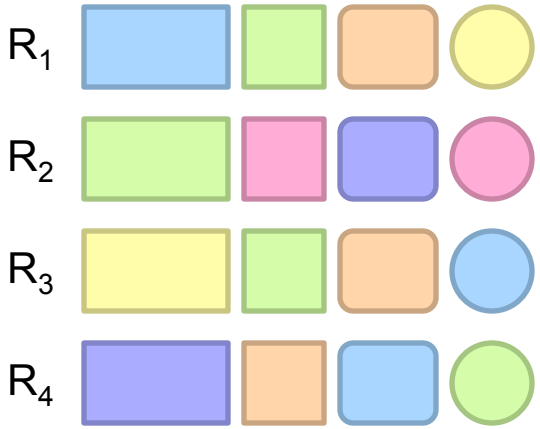
Logical

Physical

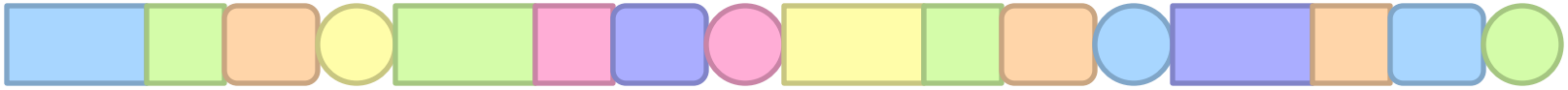


How bytes are actually represented in storage…

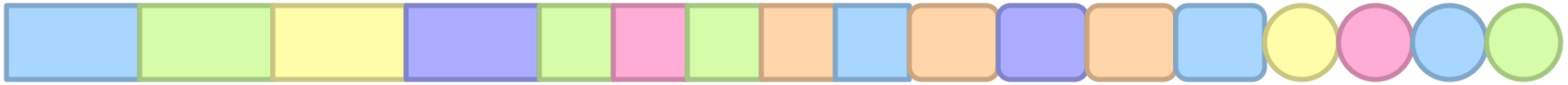
Row vs. Column Stores



Row store



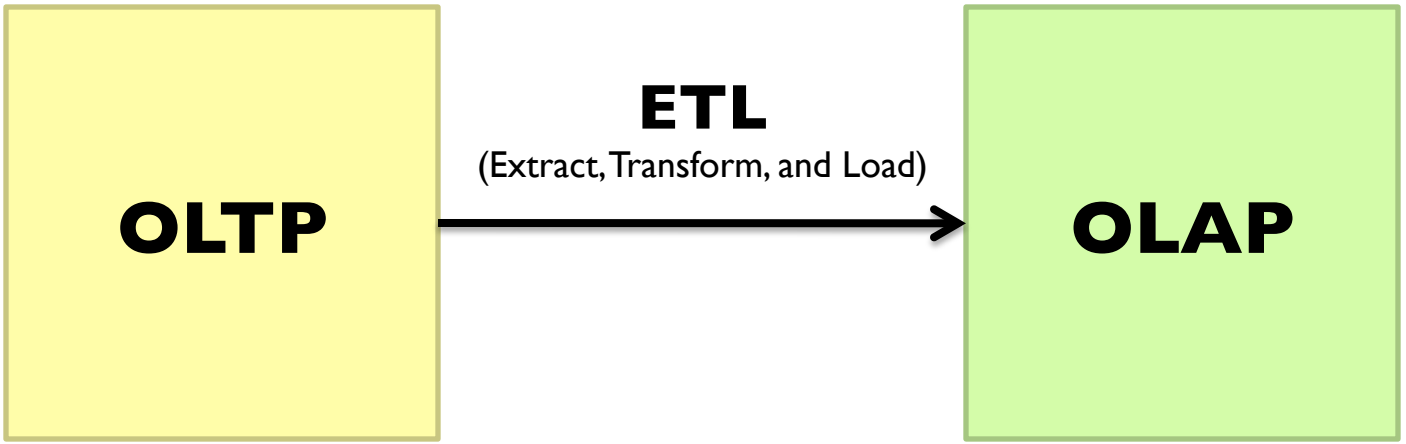
Column store



Row vs. Column Stores

- Row stores
  - Easier to modify a record: in-place updates
  - Might read unnecessary data when processing
- Column stores
  - Only read necessary data when processing
  - Tuple writes require multiple operations
  - Tuple updates are complex

OLTP/OLAP Architecture

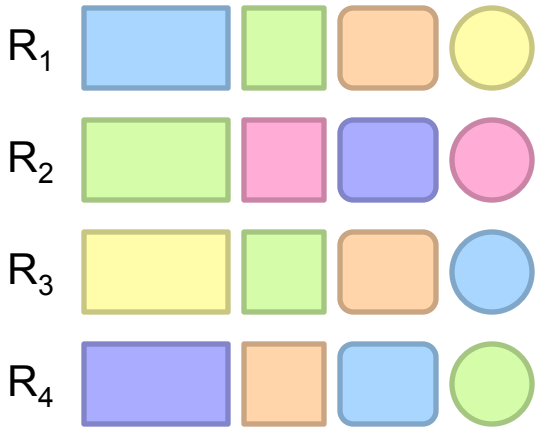


Advantages of Column Stores

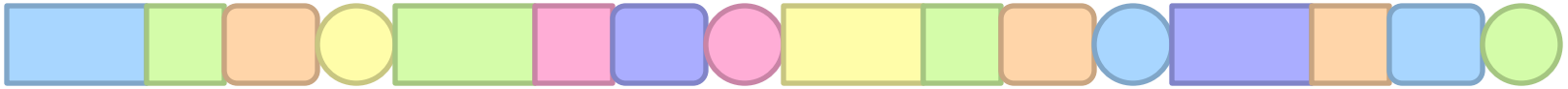
- Inherent advantages:
  - Better compression
  - Read efficiency
- Enables vectorized execution

These are well-known in traditional databases…

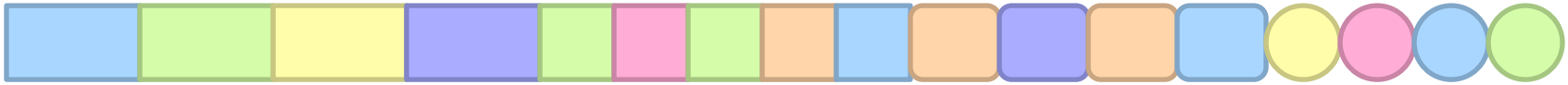
Row vs. Column Stores: Compression



Row store



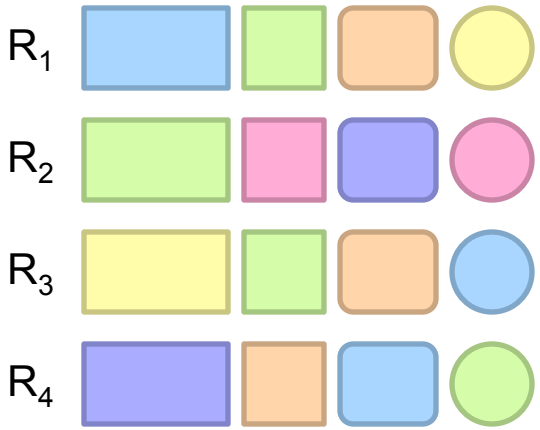
Column store



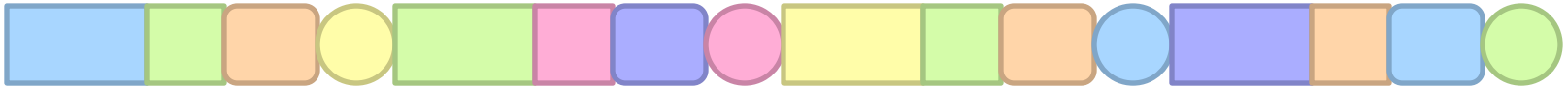
This compresses better with off-the-shelf tools, e.g., gzip.

Why?

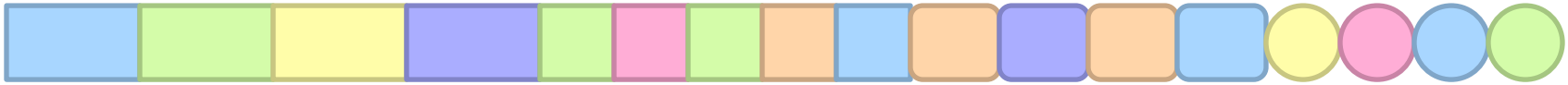
Row vs. Column Stores: Compression



Row store



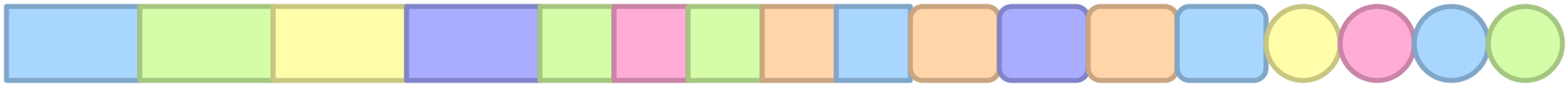
Column store



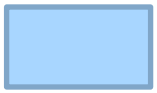
Additional opportunities for smarter compression…

Columns Stores: RLE

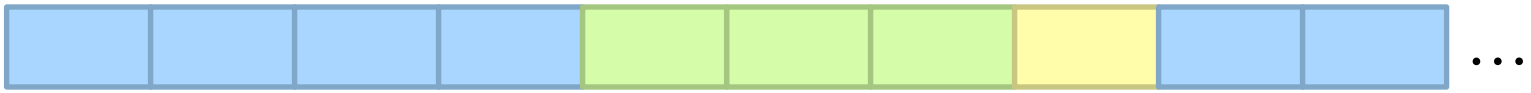
Column store



Run-length encoding example:

 is a foreign key, relatively small cardinality
(even better, boolean)

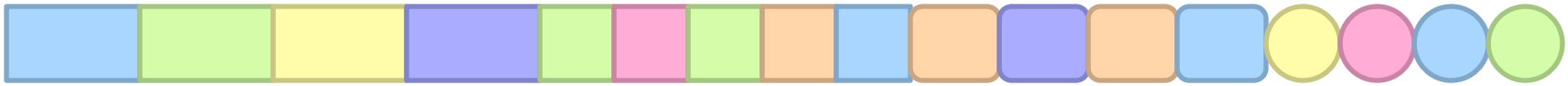
In reality:



Encode:

Columns Stores: Integer Coding

Column store

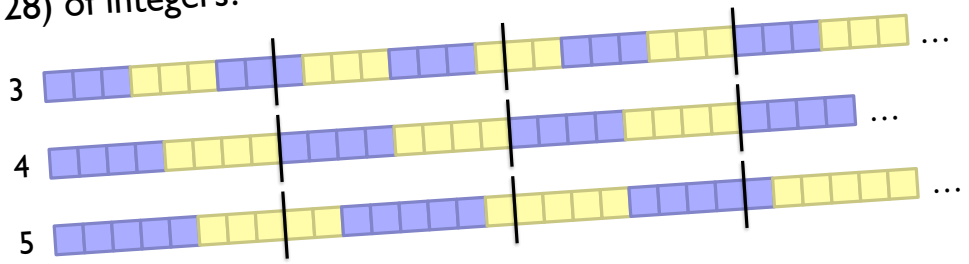


Say you're coding a bunch of integers…

Remember this?
(week 4)

Bit Packing

- What's the smallest number of bits we need to code a block (=128) of integers?



  - Efficient decompression with hard-coded decoders
- PForDelta – bit packing + separate storage of "overflow" bits

Beware of branch mispredicts?

Advantages of Column Stores

- Inherent advantages:
  - Better compression
  - Read efficiency
- Enables vectorized execution

Putting Everything Together

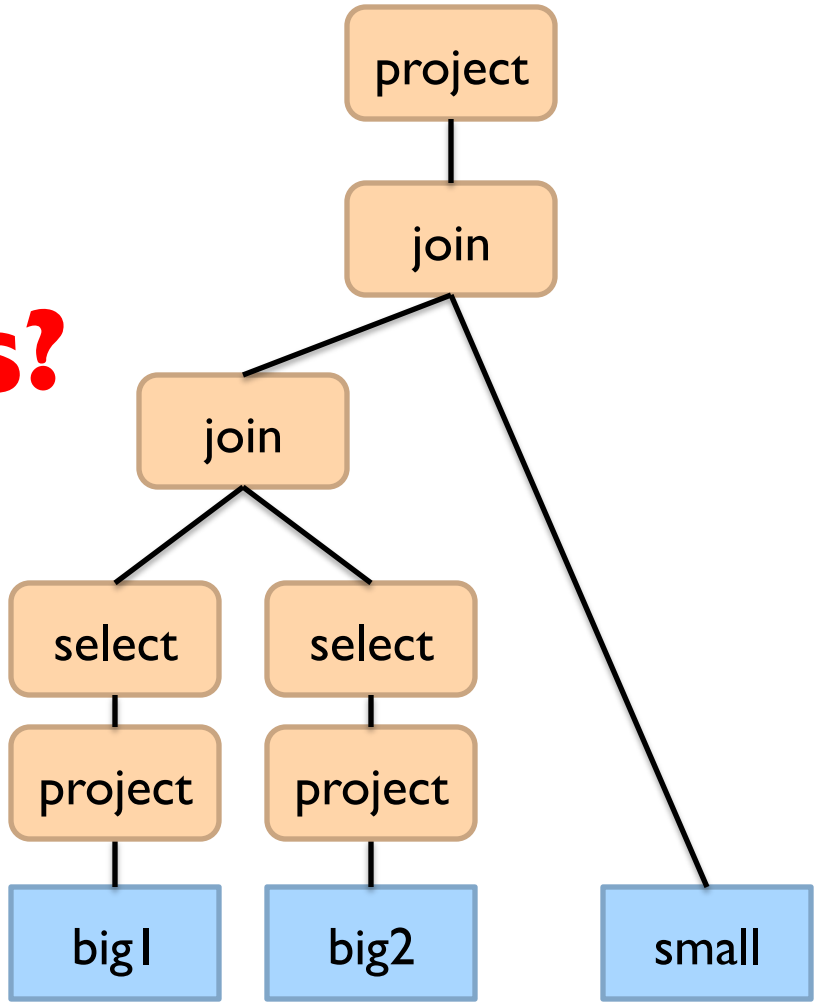
```
SELECT big1.fx, big2,fy, small,fz
FROM big1
JOIN big2 ON big1.id1 = big2.id1
JOIN small ON big1.id2 = small.id2
WHERE big1.fx = 2015 AND
      big2.f1 < 40 AND
      big2.f2 > 2;
```

Remember this?

Build logical plan

Optimize logical plan

Select physical plan



Advantages of Column Stores

- Inherent advantages:
  - Better compression
  - Read efficiency
- Enables vectorized execution

```
val size = 100000000

var col = new Array[Int](size)            // List of random ints
var selected = new Array[Boolean](size)   // Matches a predicate?
```

```
for (i <- 0 until size) {
  selected(i) = col(i) > 0
}
```

```
for (i <- 0 until size by 8) {
  selected(i) = col(i) > 0
  selected(i+1) = col(i+1) > 0
  selected(i+2) = col(i+2) > 0
  selected(i+3) = col(i+3) > 0
  selected(i+4) = col(i+4) > 0
  selected(i+5) = col(i+5) > 0
  selected(i+6) = col(i+6) > 0
  selected(i+7) = col(i+7) > 0
}
```

Which is faster?
Why?

On my laptop: 409ms
(avg over 10 trials)

On my laptop: 174ms
(avg over 10 trials)

```
val size = 100000000

var col = new Array[Int](size)            // List of random ints
var selected = new Array[Boolean](size)   // Matches a predicate?
```

```
for (i <- 0 until size) {
  selected(i) = col(i) > 0
}
```

```
for (i <- 0 until size by 8) {
  selected(i) = col(i) > 0
  selected(i+1) = col(i+1) > 0
  selected(i+2) = col(i+2) > 0
  selected(i+3) = col(i+3) > 0
  selected(i+4) = col(i+4) > 0
  selected(i+5) = col(i+5) > 0
  selected(i+6) = col(i+6) > 0
  selected(i+7) = col(i+7) > 0
}
```

Why does it matter?

```
SELECT pageURL, pageRank
FROM Rankings WHERE pageRank > X;
```

On my laptop: 409ms
(avg over 10 trials)

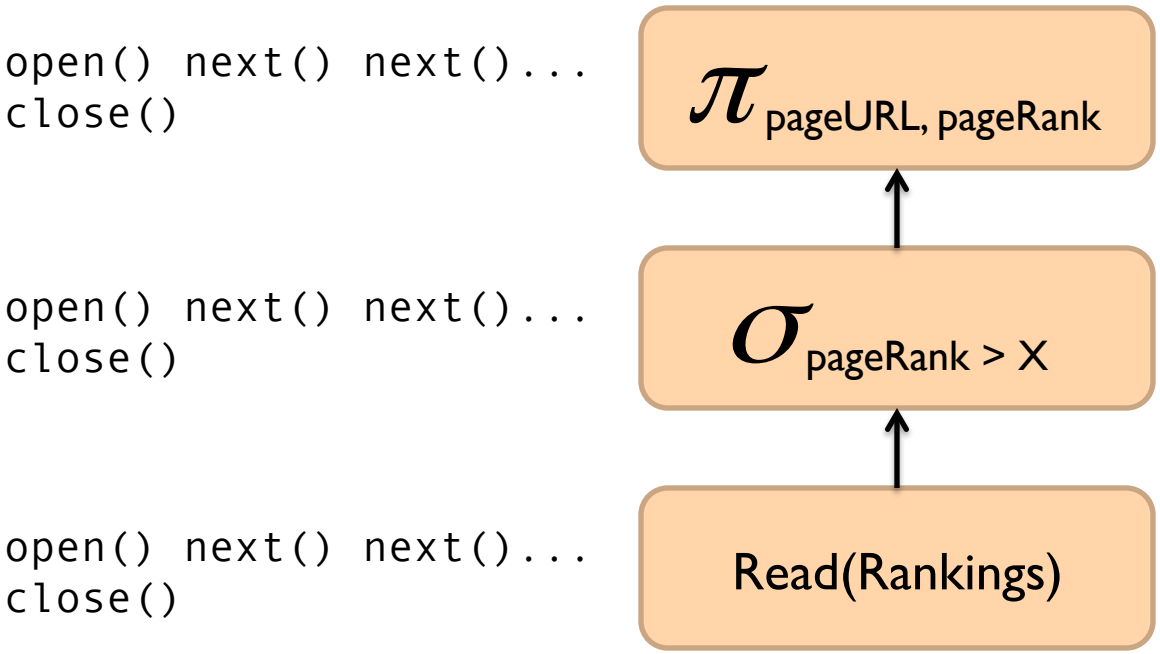
On my laptop: 174ms
(avg over 10 trials)

Actually, it's worse than that!

Each operator implements a common interface

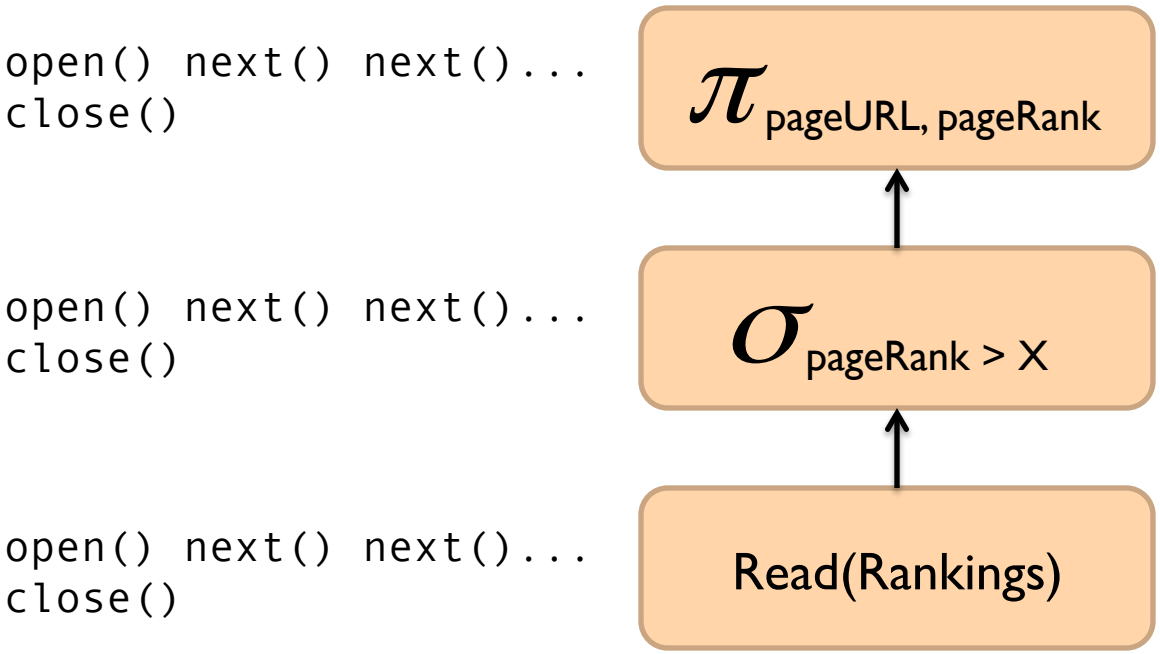
`open()`	Initialize, reset internal state, etc.
`next()`	Advance and deliver next tuple
`close()`	Clean up, free resources, etc.

Execution driven by repeated calls
to top of operator tree




```
SELECT pageURL, pageRank
FROM Rankings WHERE pageRank > X;
```

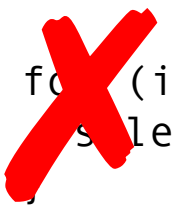
Very little actual computation is being done!




```
SELECT pageURL, pageRank
FROM Rankings WHERE pageRank > X;
```

Solution?

```
val size = 100000000

var col = new Array[Int](size)            // List of random ints
var selected = new Array[Boolean](size)   // Matches a predicate?
```



```
for (i <- 0 until size) {
  selected(i) = col(i) > 0
}
```

```
for (i <- 0 until size by 8) {
  selected(i) = col(i) > 0
  selected(i+1) = col(i+1) > 0
  selected(i+2) = col(i+2) > 0
  selected(i+3) = col(i+3) > 0
  selected(i+4) = col(i+4) > 0
  selected(i+5) = col(i+5) > 0
  selected(i+6) = col(i+6) > 0
  selected(i+7) = col(i+7) > 0
}
```

Vectorized Execution

`next()` returns a vector of tuples

All operators rewritten to work on vectors of tuples

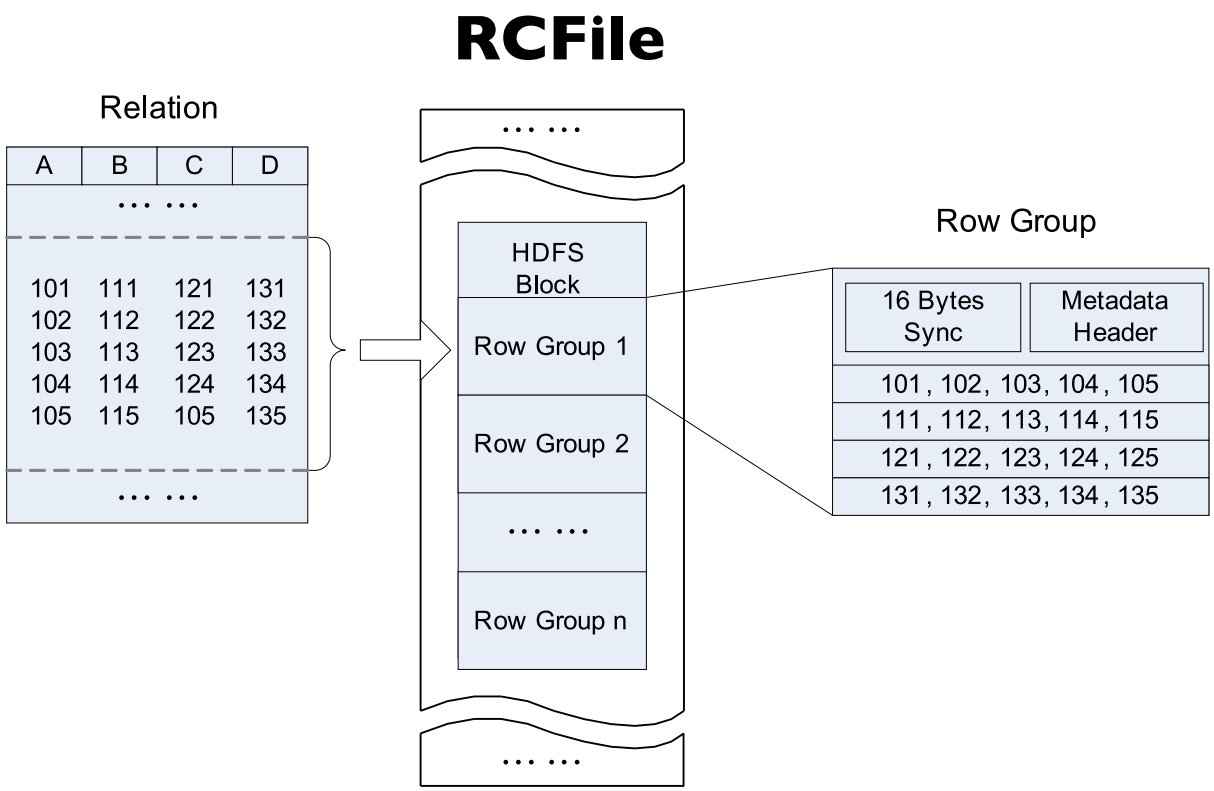
Advantages of Column Stores

- Inherent advantages:
  - Better compression
  - Read efficiency
- Enables vectorized execution

These are well-known in traditional databases…

Why not in Hadoop?

Why not in Hadoop? No reason why not!



Source: He et al. (2011) RCFile: A Fast and Space-Efficient Data Placement Structure in MapReduce-based Warehouse Systems. ICDE.

Vectorized Execution?

```
set hive.vectorized.execution.enabled = true;
```

Batch of rows, organized as columns:

```
class VectorizedRowBatch {
  boolean selectedInUse;
  int[] selected;
  int size;
  ColumnVector[] columns;
}

class LongColumnVector extends ColumnVector {
  long[] vector
}
```

Vectorized Execution


```
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;
      }
    }
  }
}
```

Vectorized operator example

Additional (Related) Trick


```
SELECT x, y
FROM z WHERE x * (1 - y)/100 < 434;
```

Predicate is "interpreted" as

```
LessThan(
  Multiply(Attribute("x"),
    Divide(Minus(Literal("1"), Attribute("y")), 100)),
  434)
```

Slow!

Dynamic code generation
(feed AST into Scala compiler to generate bytecode):

```
row.get("x") * (1 - row.get("y"))/100 < 434
```

Much faster!

What about semi-structured data?

```
message AddressBook {
  required string owner;
  repeated string ownerPhoneNumbers;
  repeated group contacts {
    required string name;
    optional string phoneNumber;
  }
}
```

Required: exactly one occurrence
Optional: 0 or 1 occurrence
Repeated: 0 or more occurrences

Columnar Decomposition

Column	Type
owner	string
ownerPhoneNumbers	string
contacts.name	string
contacts.phoneNumber	string

What's the issue?

What's the solution?

- Google's Dremel storage model
- Open-source implementation in Parquet

Source: https://blog.twitter.com/2013/dremel-made-simple-with-parquet

Optional and Repeated Elements

Schema: List of Strings	**Data:** ["a", "b", "c", ...]

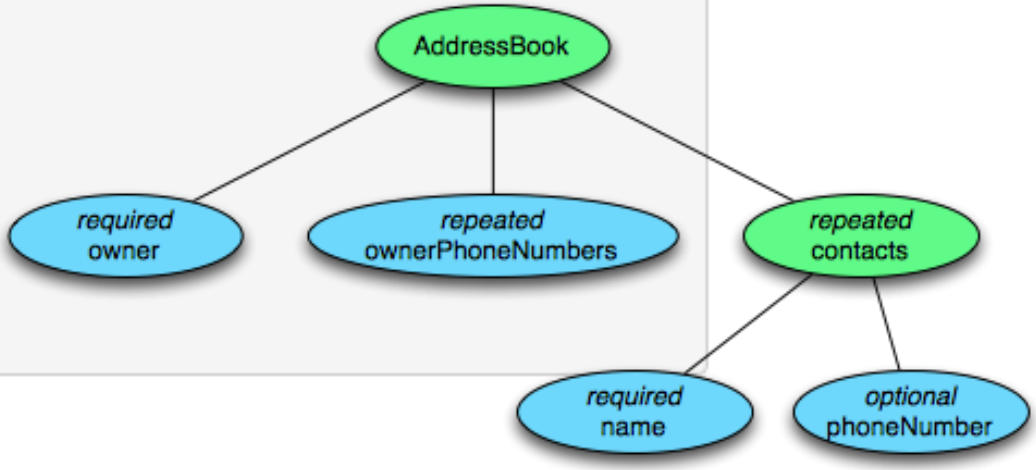
message ExampleList {
  repeated string list;
}
	{
	  list: "a",
	  list: "b",
	  list: "c",
	  …
	}

Schema: Map of strings to strings	**Data:** {"AL" => "Alabama", ...}

message ExampleMap {
  repeated group map {
  required string key;
  optional string value;
  }
}
	{
	  map: {
	  key: "AL",
	  value: "Alabama"
	  },
	  map: {
	  key: "AK",
	  value: "Alaska"
	  },
	  …
	}

Tree Decomposition

```
message AddressBook {
  required string owner;
  repeated string ownerPhoneNumbers;
  repeated group contacts {
    required string name;
    optional string phoneNumber;
  }
}
```



Columnar Decomposition

Column	Type
owner	string
ownerPhoneNumbers	string
contacts.name	string
contacts.phoneNumber	string

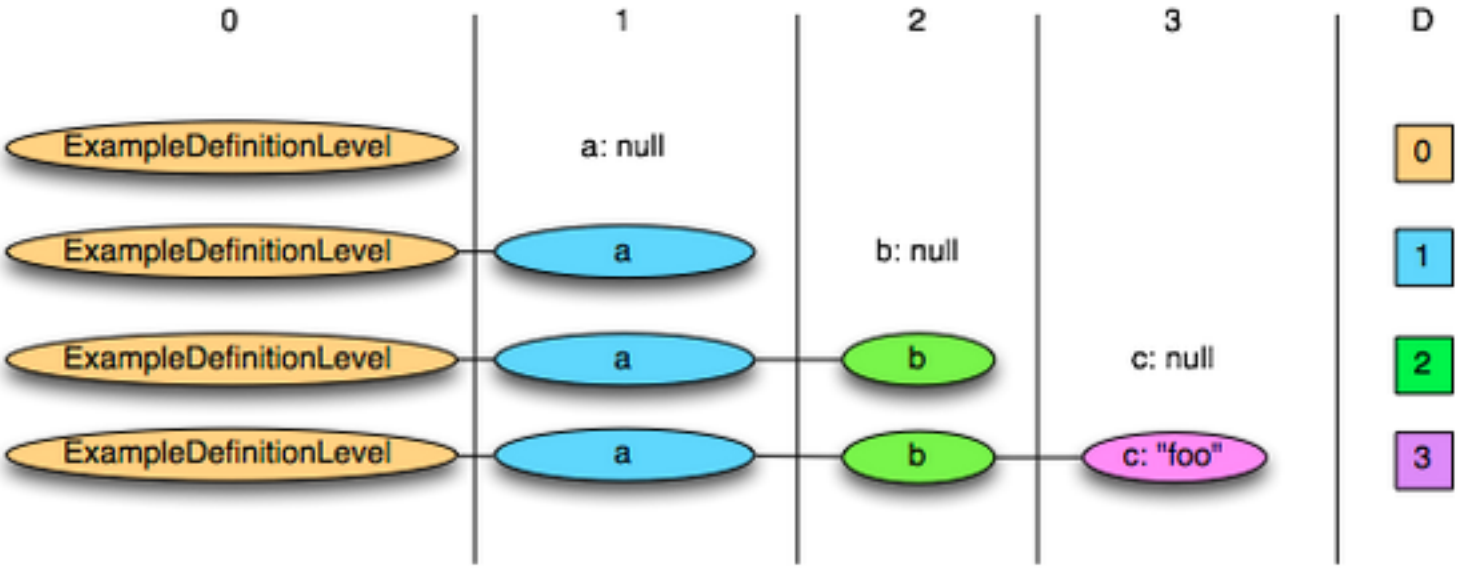
What other information do we need to store?

Definition Level

```
message ExampleDefinitionLevel {
  optional group a {
    optional group b {
      optional string c;
    }
  }
}
```

Value	Definition Level
a: null	0
a: { b: null }	1
a: { b: { c: null } }	2
a: { b: { c: "foo" } }	3 (actually defined)

Definition Level: Illustration



Repetition Level

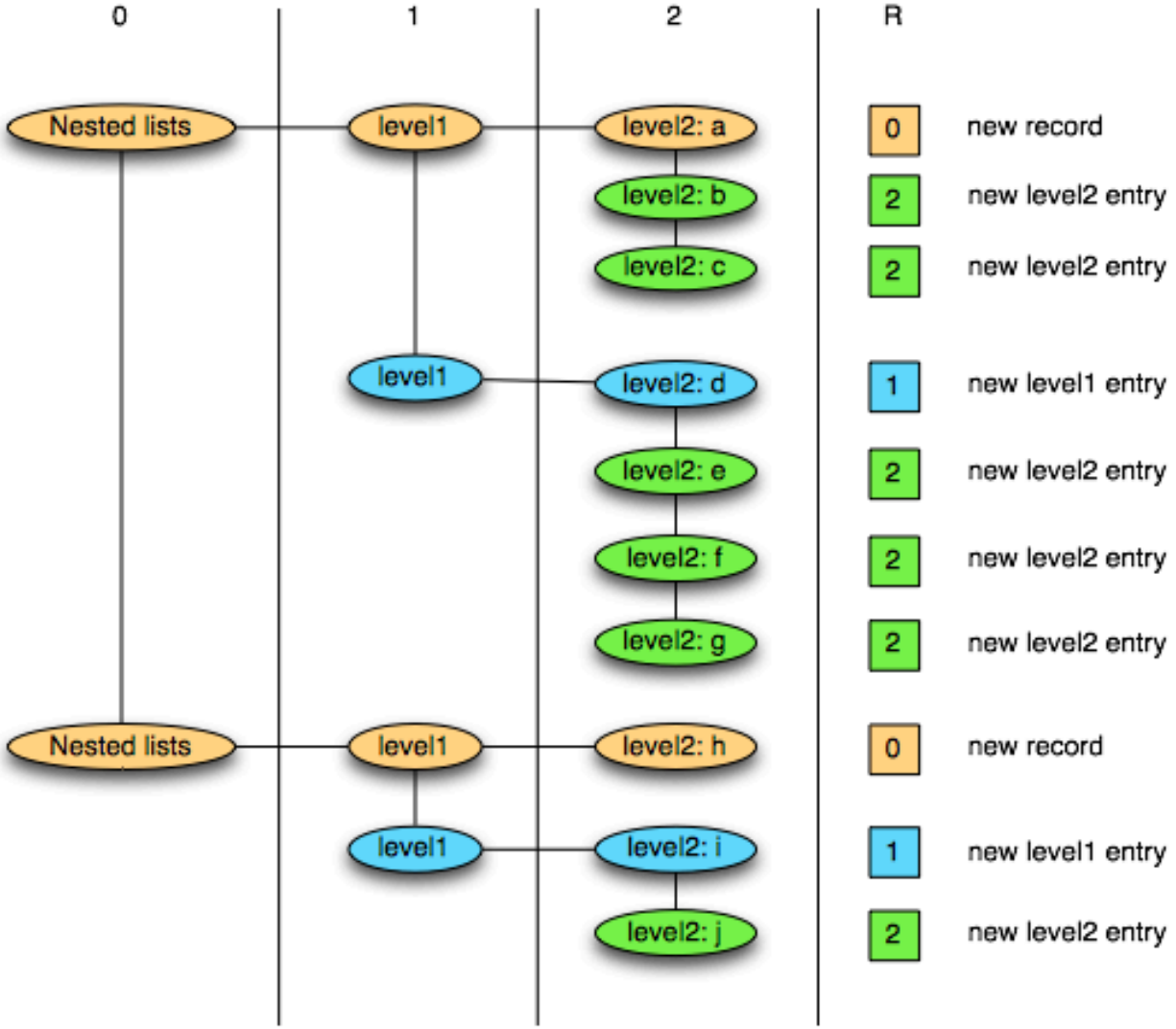
Schema:

```
message nestedLists {
   repeated group level1 {
      repeated string level2;
   }
}
```

Data: `[[a,b,c],[d,e,f,g]],[[h],[i,j]]`

```
{
      level1: {
            level2: a
            level2: b
            level2: c
      },
      level1: {
            level2: d
            level2: e
            level2: f
            level2: g
      }
}
{
      level1: {
            level2: h
      },
      level1: {
            level2: i
            level2: j
      }
}
```

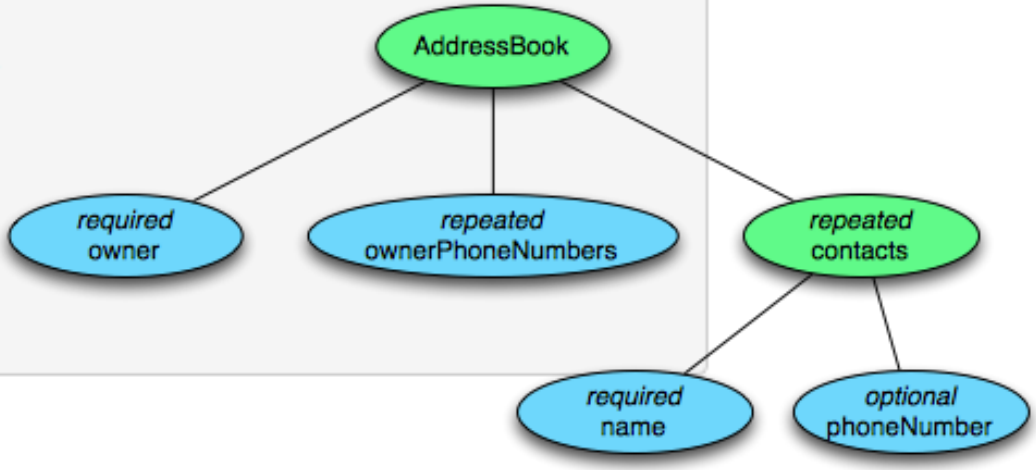
Repetition Level: Illustration



0 marks new record and implies creating a new level1 and level2 list
1 marks new level1 list and implies creating a new level2 list as well.
2 marks every new element in a level2 list.

Putting It Together

```
message AddressBook {
  required string owner;
  repeated string ownerPhoneNumbers;
  repeated group contacts {
    required string name;
    optional string phoneNumber;
  }
}
```



Columnar Decomposition

Column	Max Definition level	Max Repetition level
`owner`	0 (`owner` is *required*)	0 (no repetition)
`ownerPhoneNumbers`	1	1 (*repeated*)
`contacts.name`	1 (`name` is *required*)	1 (`contacts` is *repeated*)
`contacts.phoneNumber`	2 (`phoneNumber` is *optional*)	1 (`contacts` is *repeated*)

Sample Projection

```
AddressBook {
  owner: "Julien Le Dem",
  ownerPhoneNumbers: "555 123 4567",
  ownerPhoneNumbers: "555 666 1337",
  contacts: {
    name: "Dmitriy Ryaboy",
    phoneNumber: "555 987 6543",
  },
  contacts: {
    name: "Chris Aniszczyk"
  }
}
AddressBook {
  owner: "A. Nonymous"
}
```

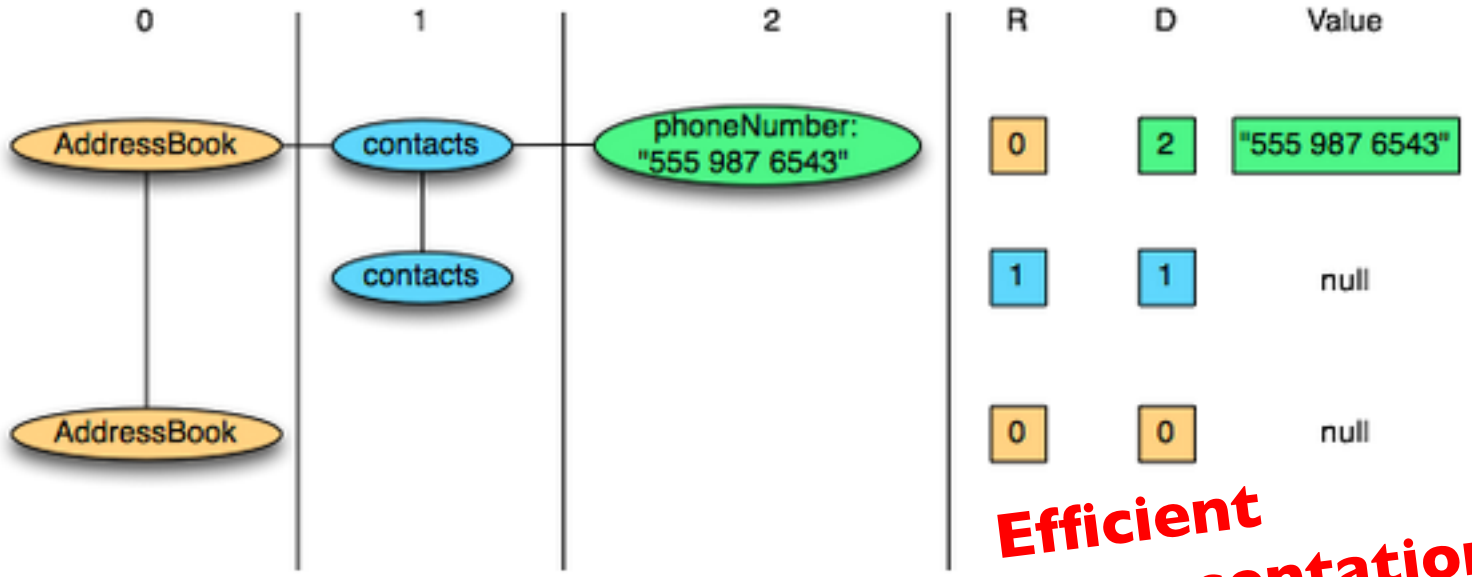
Project onto contacts.phoneNumber

```
AddressBook {
  contacts: {
    phoneNumber: "555 987 6543"
  }
  contacts: {
  }
}
AddressBook {
}
```

Physical Layout

Columnar Decomposition

Column	Type
owner	string
ownerPhoneNumbers	string
contacts.name	string
contacts.phoneNumber	string



Efficient Representations?

Key Ideas

- Binary representations are good
- Binary representations need schemas
- Schemas allow logical/physical separation
- Logical/physical separation allows you to do cool things

A Major Step Backwards?

- MapReduce is a step backward in database access:
  - Schemas are good
  - Separation of the schema from the application is good
  - High-level access languages are good
- MapReduce is poor implementation
  - Brute force and only brute force (no indexes, for example)
- MapReduce is not novel
- MapReduce is missing features
  - Bulk loader, indexing, updates, transactions…
- MapReduce is incompatible with DMBS tools

Source: Blog post by DeWitt and Stonebraker

Source: Wikipedia (Card Catalog)

Hadoop + Full-Text Indexes

```
status = load '/tables/statuses/2011/03/01'
    using StatusProtobufPigLoader()
    as (id: long, user_id: long, text: chararray, ...);

filtered = filter status by text matches '.*\\bhadoop\\b.*';
…
```

Pig performs a brute force scan

Then promptly chucks out most of the data Stupid.

Source: Lin et al. (2011) Full-Text Indexing for Optimizing Selection Operations in Large-Scale Data Analytics. MAPREDUCE Workshop.

"Trying to find a needle in a haystack… with a snowplow"
@squarecog

Hadoop + Full-Text Indexes

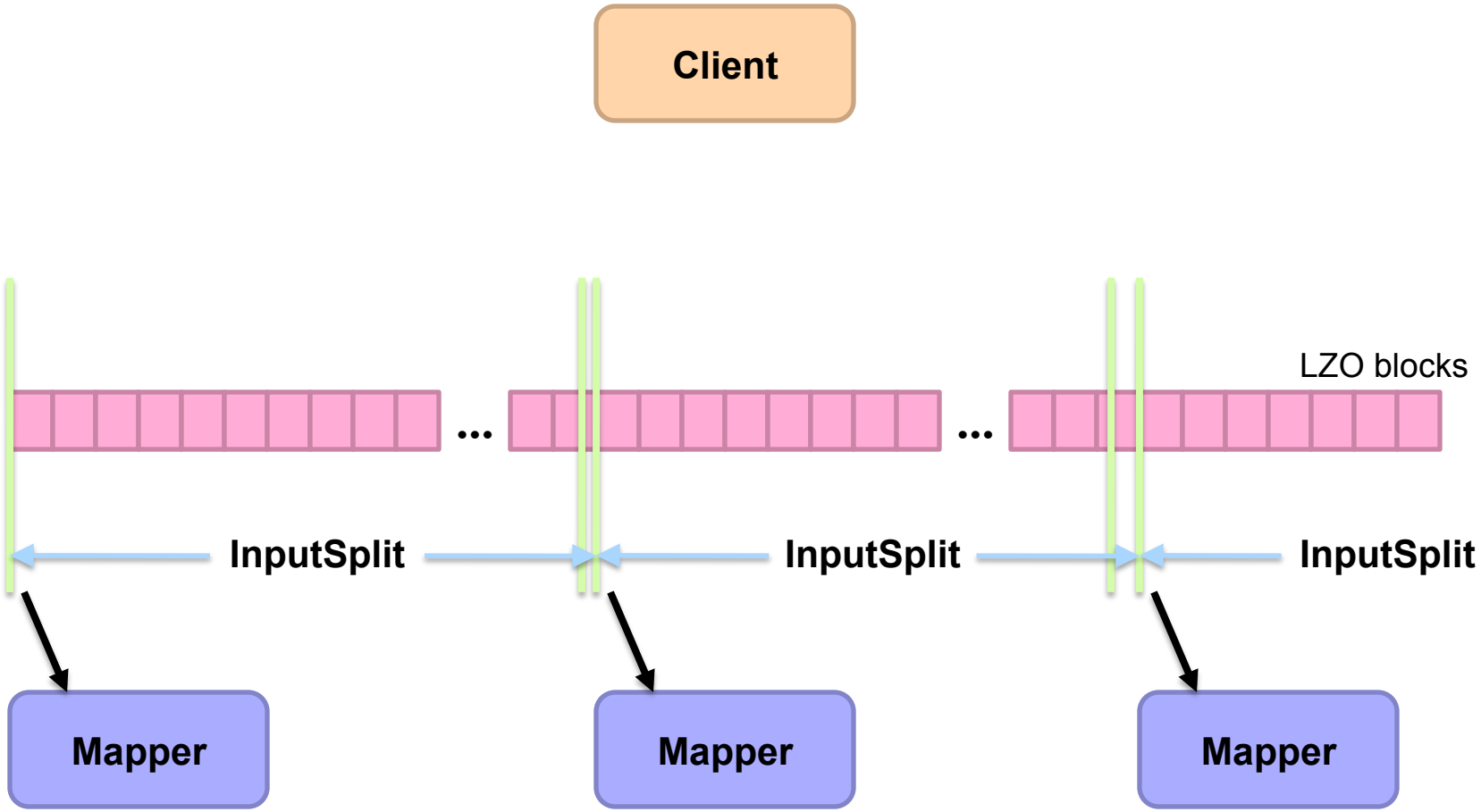
```
status = load '/tables/statuses/2011/03/01'
    using StatusProtobufPigLoader()
    as (id: long, user_id: long, text: chararray, ...);

filtered = filter status by text matches '.*\\bhadoop\\b.*';
…
```

Pig performs a brute force scan

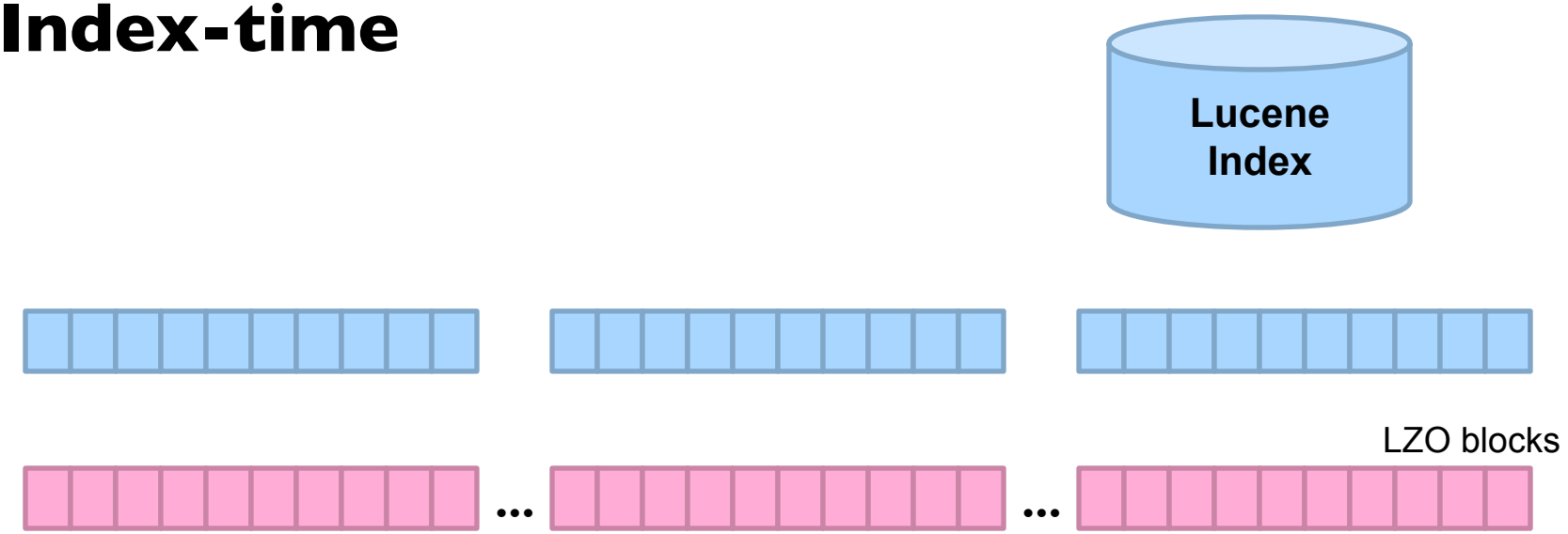
Then promptly chucks out most of the data Stupid.

Uhhh… how about an index?

Use Lucene full-text index


Client
LZO blocks
InputSplit
InputSplit
InputSplit
Mapper
Mapper
Mapper

Index-time

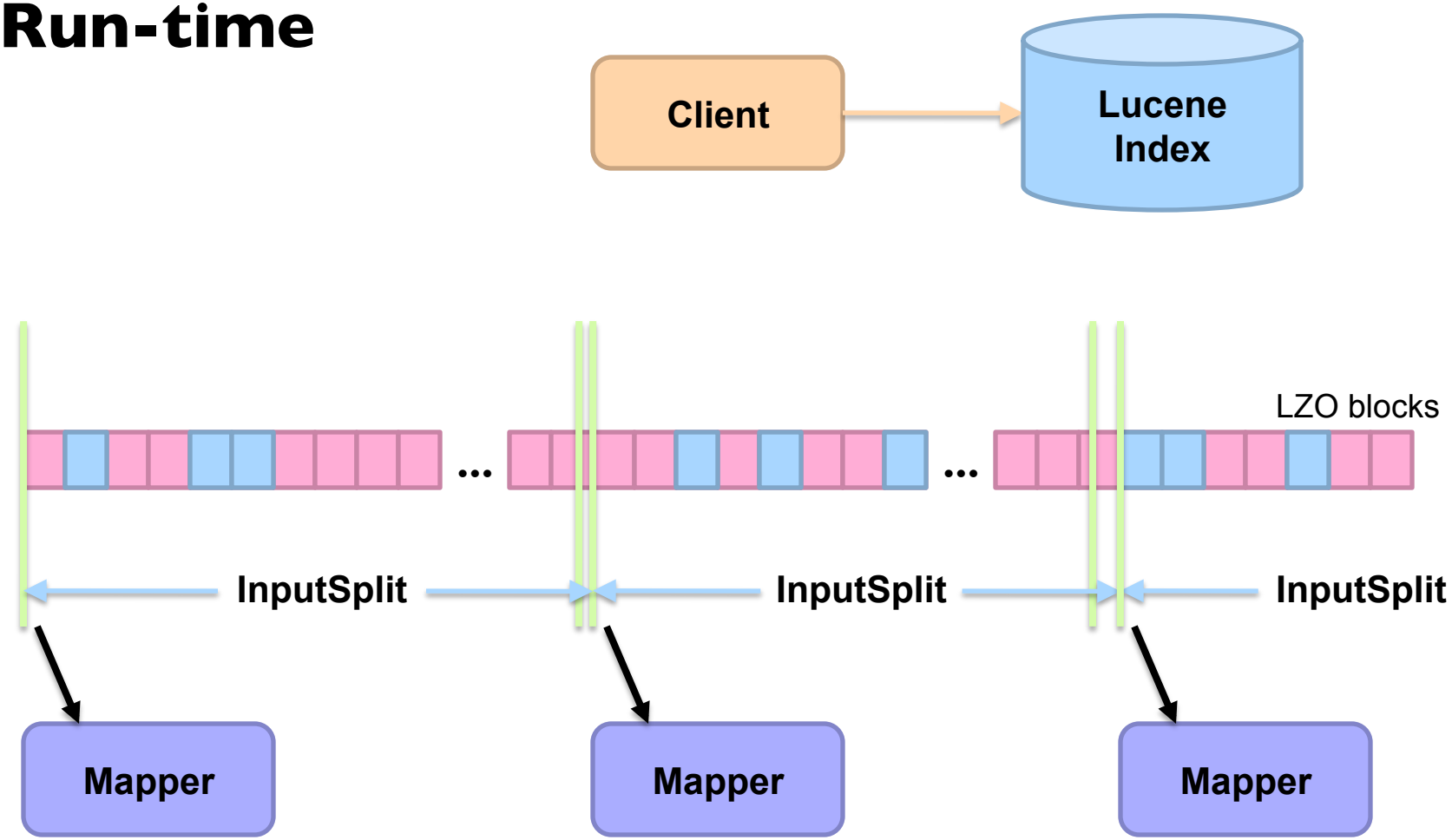


Index for selection on tweet content

Build "pseudo-document" for each Lzo block

Index pseudo-documents with Lucene

Run-time



Only process blocks known to satisfy selection criteria

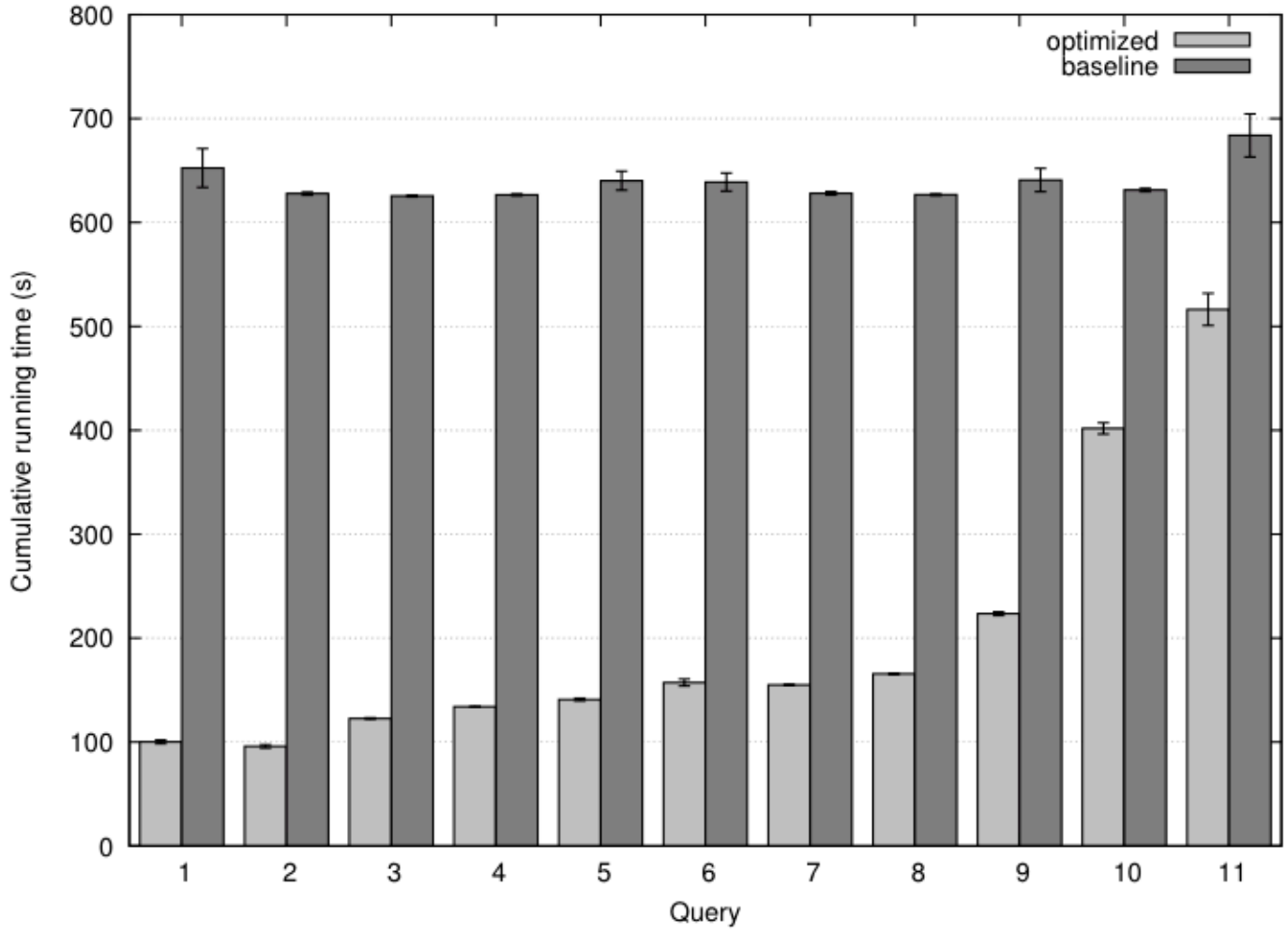
Hadoop Integration

- Everything encapsulated in the InputFormat
- RecordReaders know what blocks to process and skip
- Completely transparent to mappers

Experiments

- Selection on tweet content
- Varied selectivity range
- One day sample data (70m tweets, 8/1/2010)

	Query	Blocks	Records	Selectivity
1	hadoop	97	105	1.517×10^{-6}
2	replication	140	151	2.182×10^{-6}
3	buffer	500	559	8.076×10^{-6}
4	transactions	819	867	1.253×10^{-5}
5	parallel	999	1159	1.674×10^{-5}
6	ibm	1437	1569	2.267×10^{-5}
7	mysql	1511	1664	2.404×10^{-5}
8	oracle	1822	1911	2.761×10^{-5}
9	database	3759	3981	5.752×10^{-5}
10	microsoft	13089	17408	2.515×10^{-4}
11	data	20087	30145	4.355×10^{-4}


optimized
baseline
Cumulative running time (s)
0
100
200
300
400
500
600
700
800
1
2
3
4
5
6
7
8
9
10
11
Query

Analytical model

- Task: prediction LZO blocks scanned by selectivity
- Poisson model: P(observing *k* occurrences in a block)

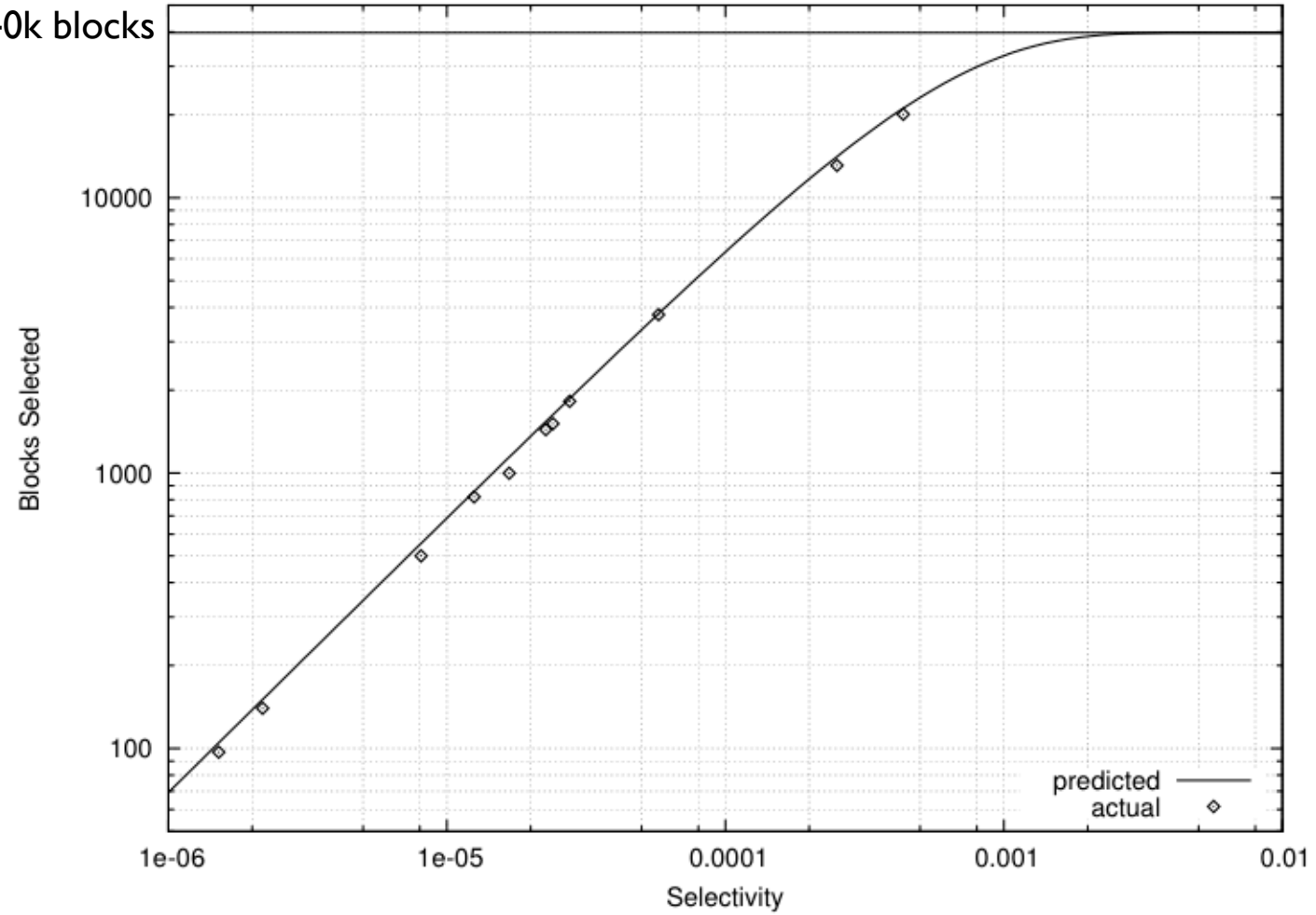
$$f(k;\lambda) = \frac{\lambda^k e^{-\lambda}}{k!} \qquad \lambda : \text{expected number of occurrences within block}$$

- E(fraction of blocks scanned):

$$1 - f(k = 0;\lambda)$$

Selectivity 0.001 → 82% of all blocks
Selectivity 0.002 → 97% of all blocks

Total: ~40k blocks



But: can predict *a priori*!

A Major Step Backwards?

- MapReduce is a step backward in database access:
  - Schemas are good
  - Separation of the schema from the application is good
  - High-level access languages are good
- MapReduce is poor implementation
  - Brute force and only brute force (no indexes, for example)
- MapReduce is not novel
- MapReduce is missing features
  - Bulk loader, indexing, updates, transactions…
- MapReduce is incompatible with DMBS tools

Source: Blog post by DeWitt and Stonebraker

Source: Wikipedia (Japanese rock garden)