



Big Data Infrastructure

CS 489/698 Big Data Infrastructure (Winter 2017)

Week 7: Analyzing Relational Data (3/3)

February 14, 2017

Jimmy Lin

David R. Cheriton School of Computer Science
University of Waterloo

These slides are available at <http://lintool.github.io/bigdata-2017w/>



This work is licensed under a Creative Commons Attribution-Noncommercial-Share Alike 3.0 United States
See <http://creativecommons.org/licenses/by-nc-sa/3.0/us/> for details

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 DBMS tools

Hadoop vs. Databases: Grep

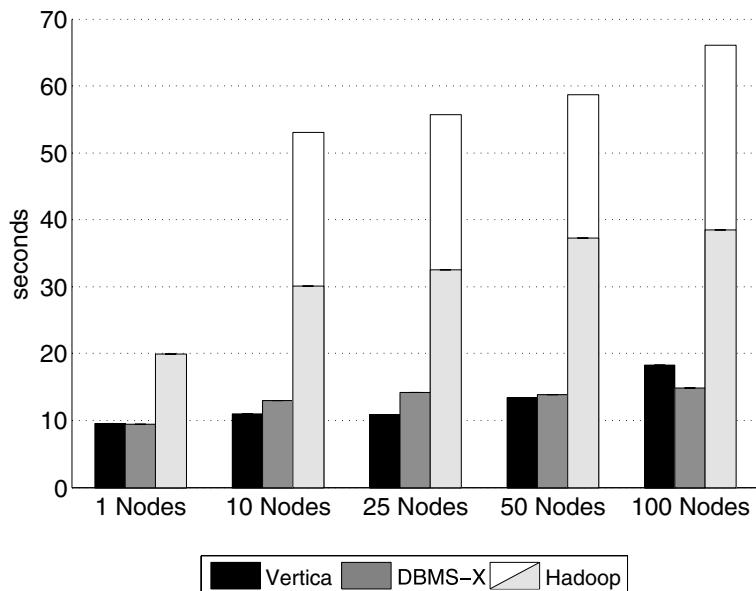


Figure 4: Grep Task Results – 535MB/node Data Set

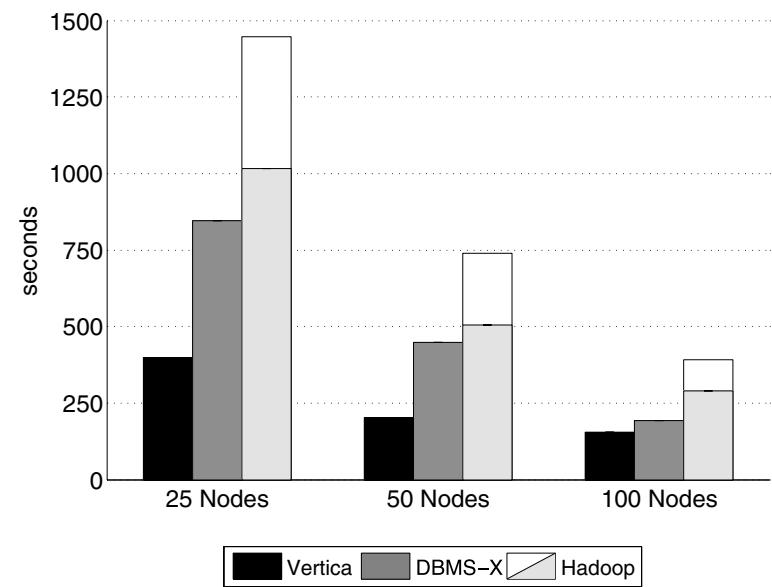


Figure 5: Grep Task Results – 1TB/cluster Data Set

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

Hadoop vs. Databases: Select

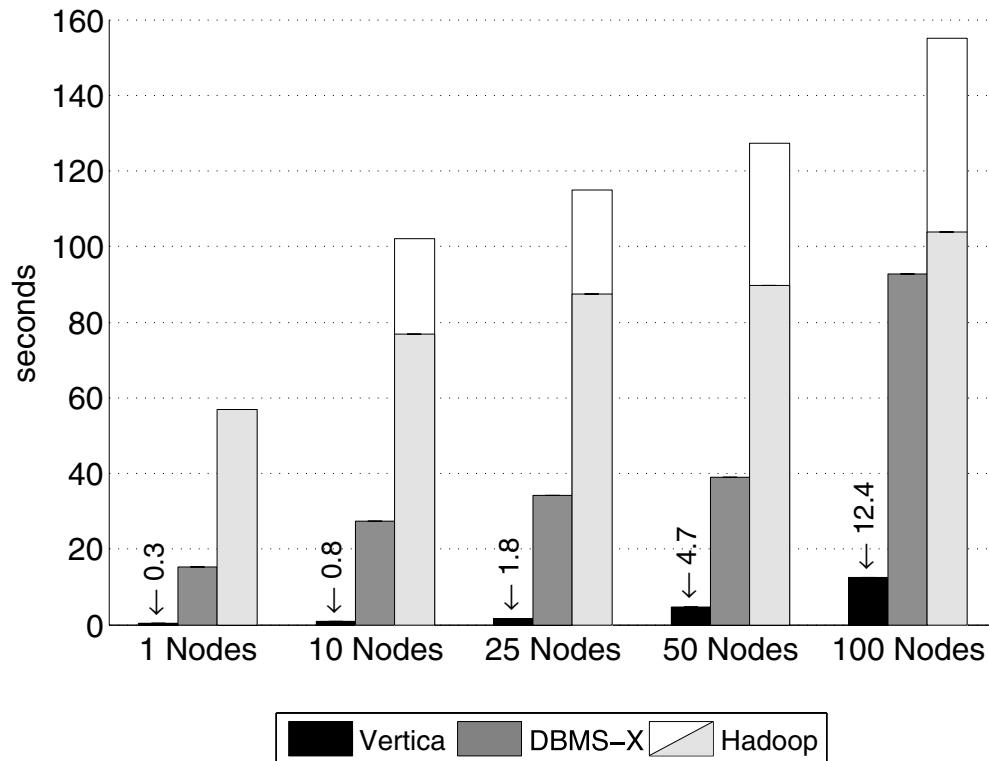


Figure 6: Selection Task Results

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

Hadoop vs. Databases: Aggregation

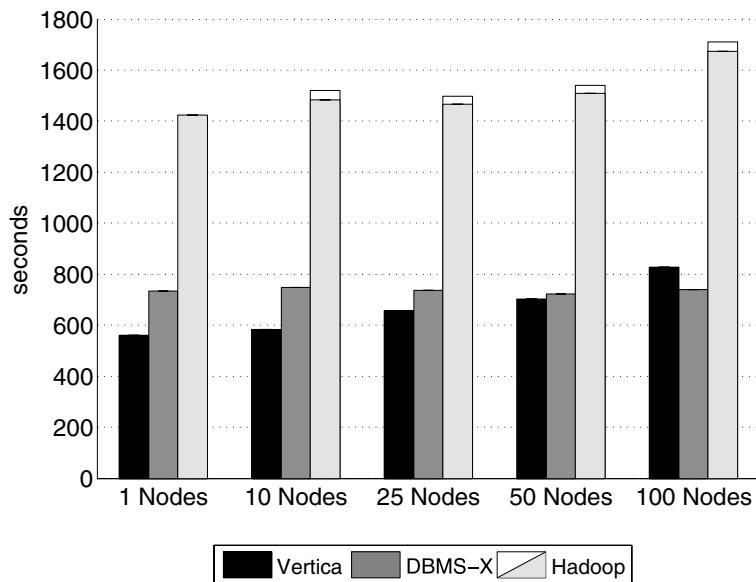


Figure 7: Aggregation Task Results (2.5 million Groups)

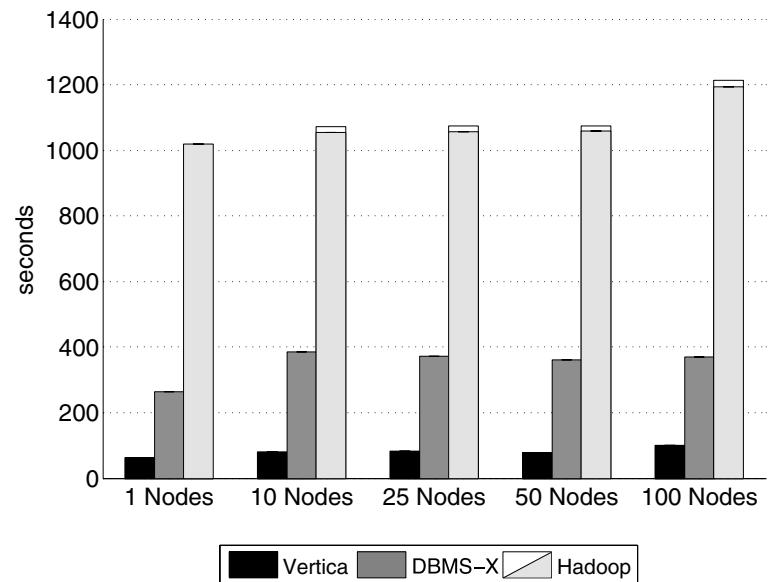


Figure 8: Aggregation Task Results (2,000 Groups)

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

Hadoop vs. Databases: Join

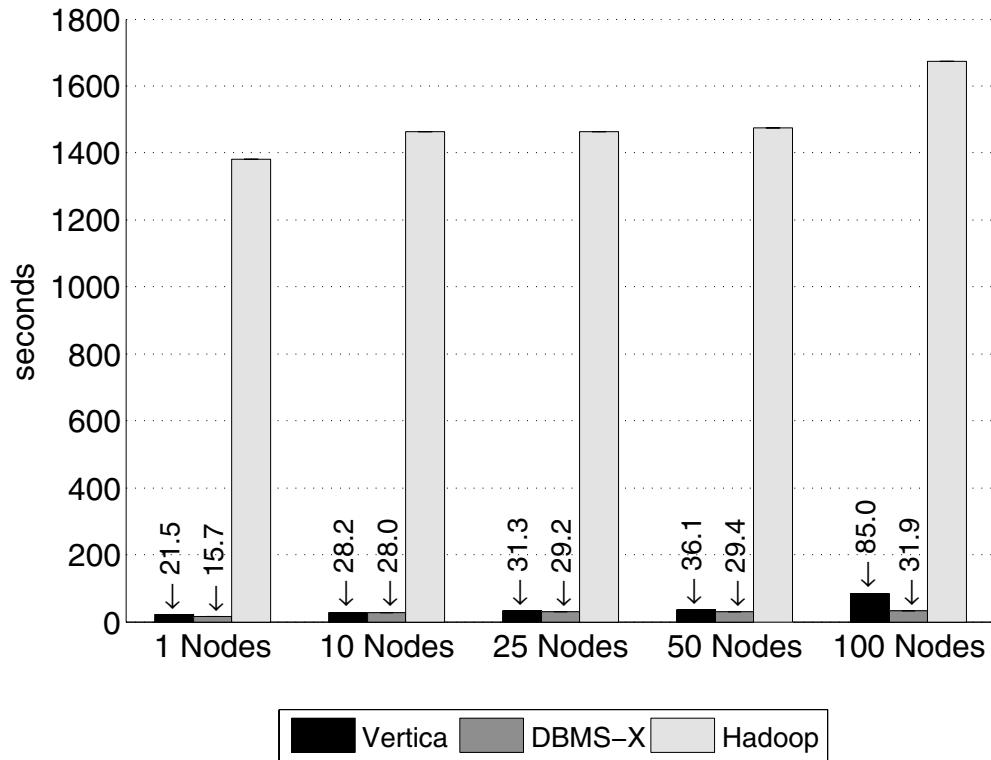


Figure 9: Join Task Results

```
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;
```

Hadoop is slow...



A close-up photograph of a large group of koi fish swimming in water. The fish are of various colors, including orange, red, black, white, and yellow. They are densely packed, creating a vibrant and textured scene. The water is slightly choppy, with small ripples around the fish.

Something seems fishy...

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

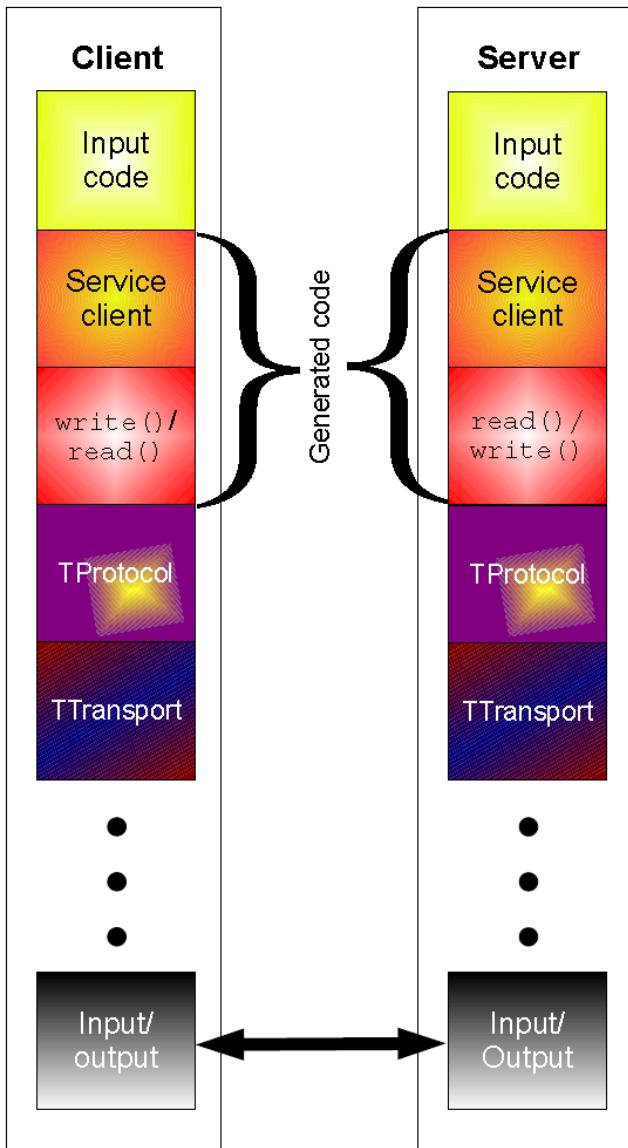
Fields can be marked as optional or required

Compiler automatically generates code for manipulating messages

Provides RPC mechanisms for service definitions

Don't like Thrift? 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;  
}
```

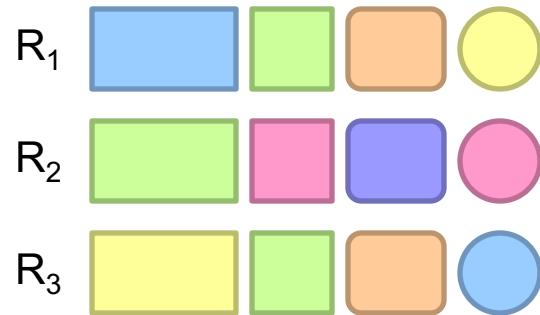
Why not...

XML or JSON?

REST?

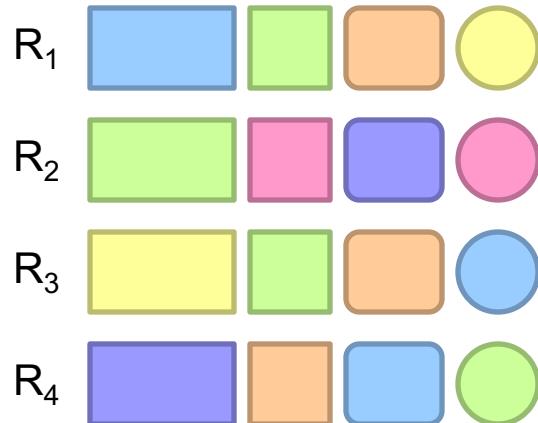
Logical

Physical



**How bytes are actually
represented in storage...**

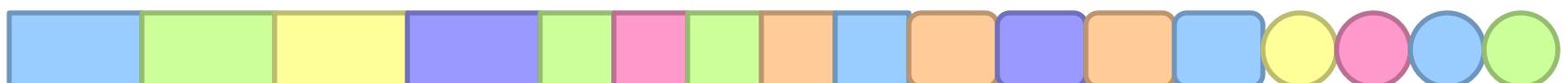
Row vs. Column Stores



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

external APIs

users

users

Frontend

Frontend

Frontend

Backend

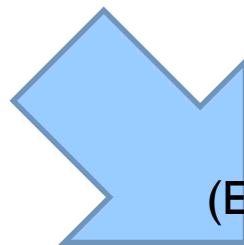
Backend

Backend

OLTP
database

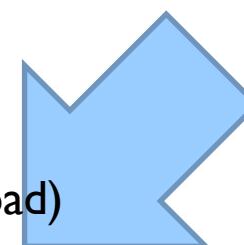
OLTP
database

OLTP
database



ETL

(Extract, Transform, and Load)



OLAP Data
Warehouse

BI tools

analysts

Advantages of Column Stores

Inherent advantages:

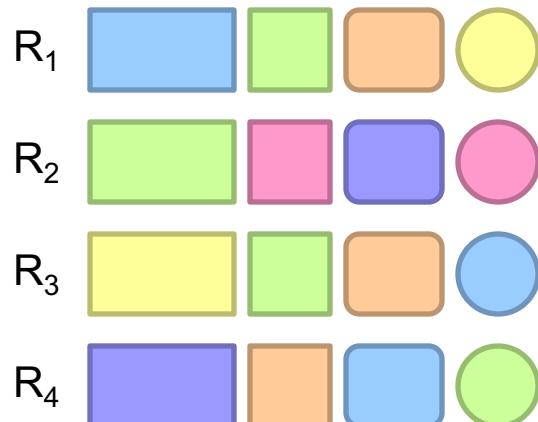
- Better compression
- Read efficiency

Works well with:

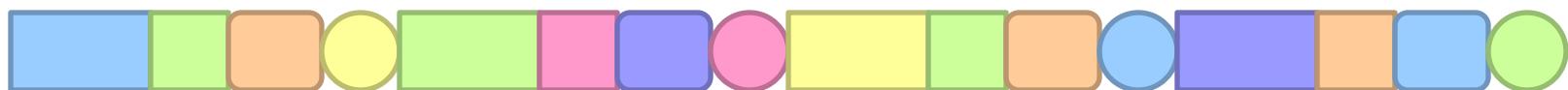
- Vectorized Execution
- Compiled Queries

These are well-known in traditional databases...

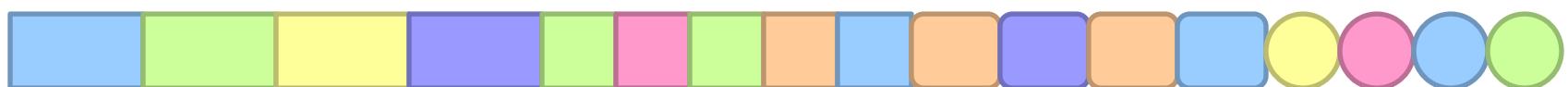
Row vs. Column Stores: Compression



Row store



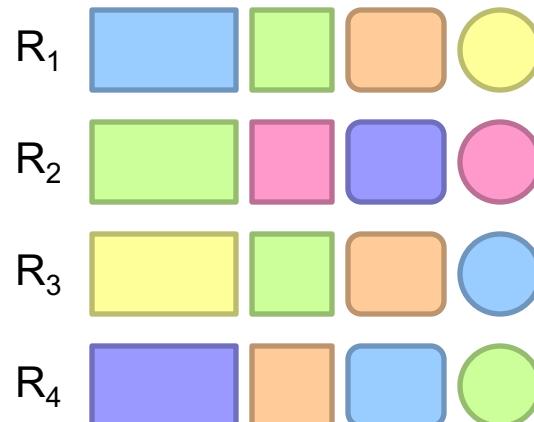
Column store



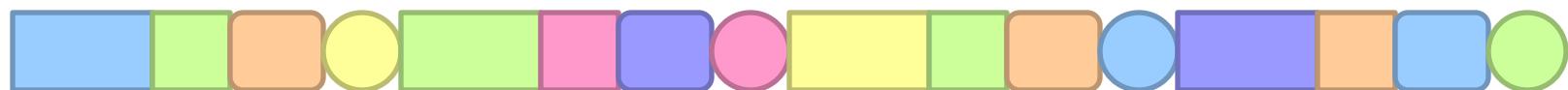
This compresses better with
off-the-shelf tools, e.g., gzip.

Why?

Row vs. Column Stores: Compression



Row store



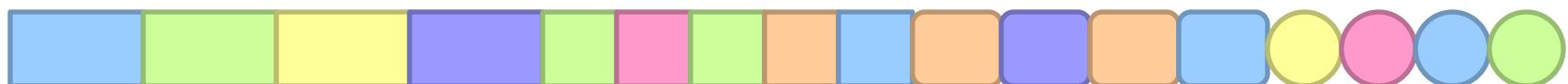
Column store



Additional opportunities for smarter compression...

Columns Stores: RLE

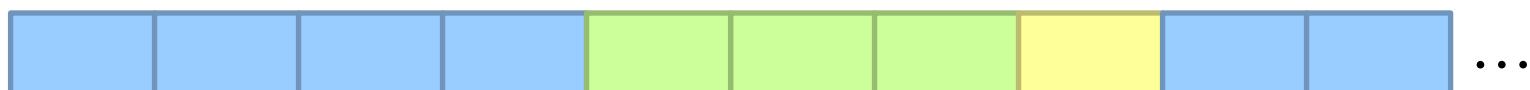
Column store



Run-length encoding example:

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

In reality:

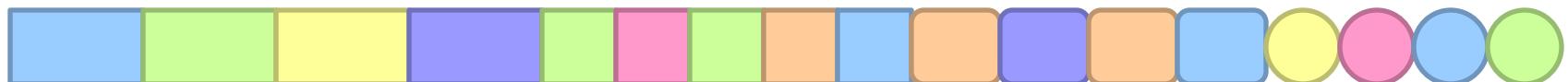


Encode:

3 2 1 ...

Columns Stores: Integer Coding

Column store



Say you're coding a bunch of integers...

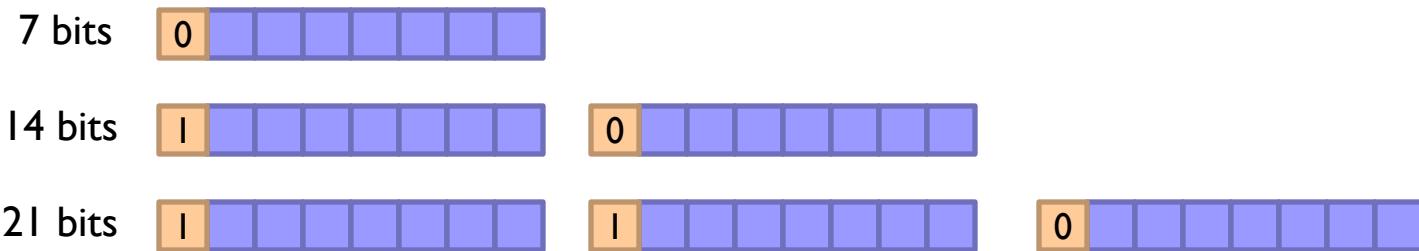
Remember this!
(week 4)

VByte

Simple idea: use only as many bytes as needed

Need to reserve one bit per byte as the “continuation bit”

Use remaining bits for encoding value



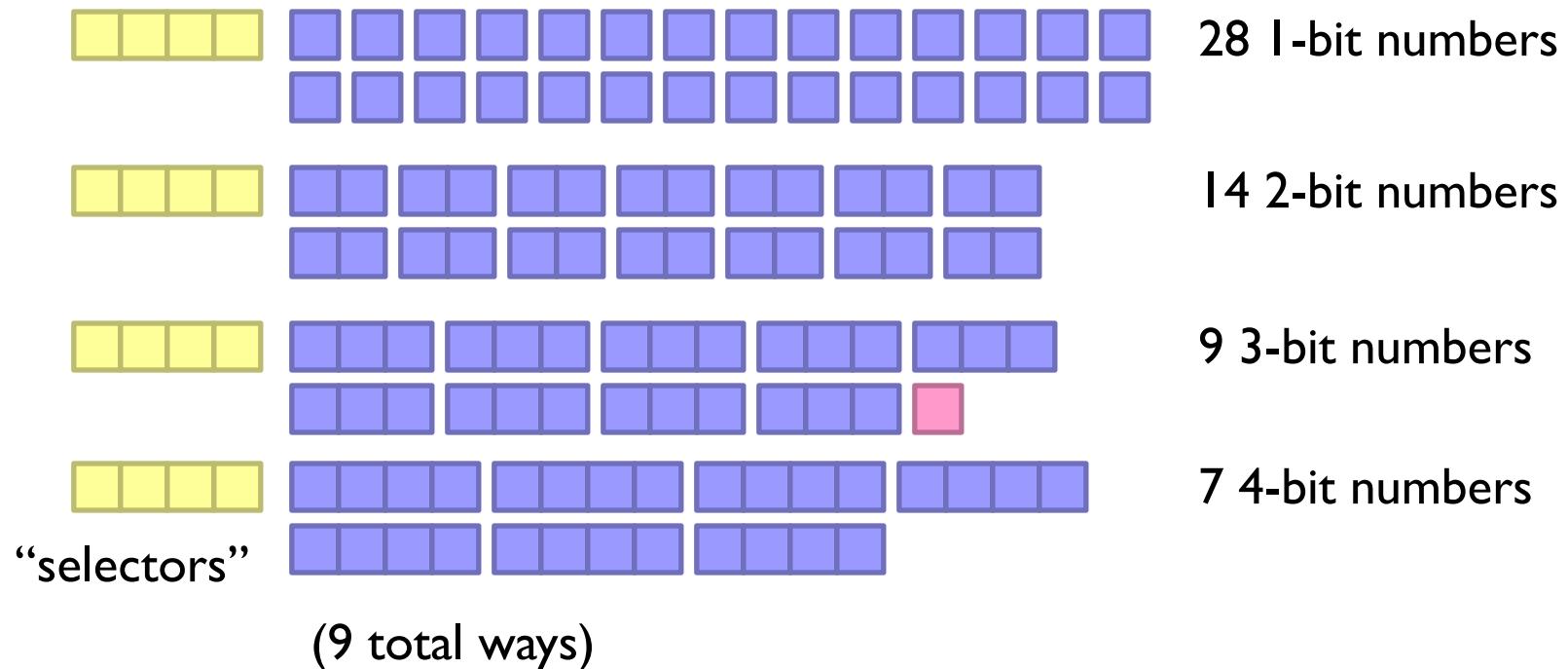
Works okay, easy to implement...

Beware of branch mispredicts!

Remember this!
(week 4)

Simple-9

How many different ways can we divide up 28 bits?



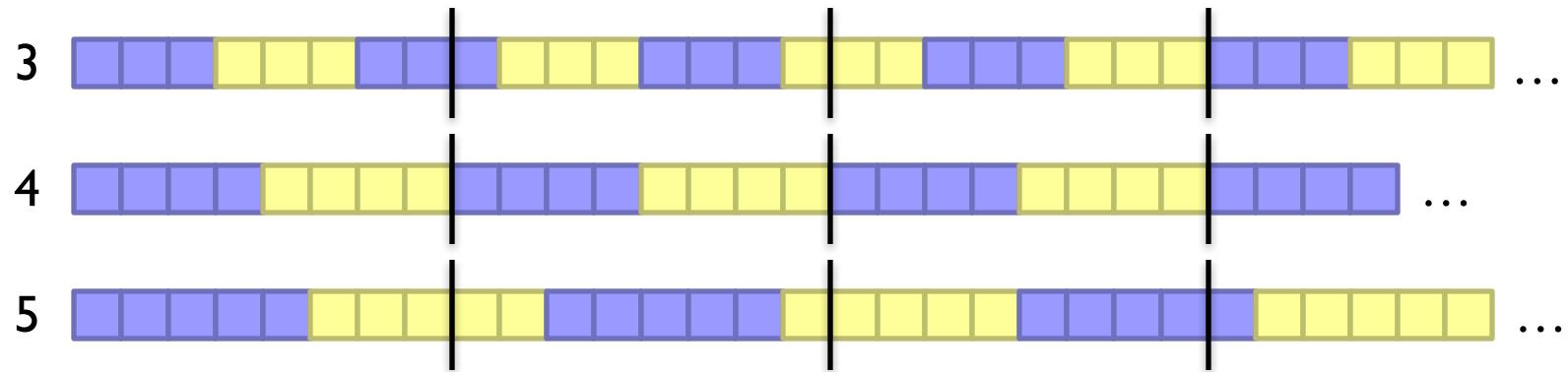
Efficient decompression with hard-coded decoders
Simple Family – general idea applies to 64-bit words, etc.

Beware of branch mispredicts?

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

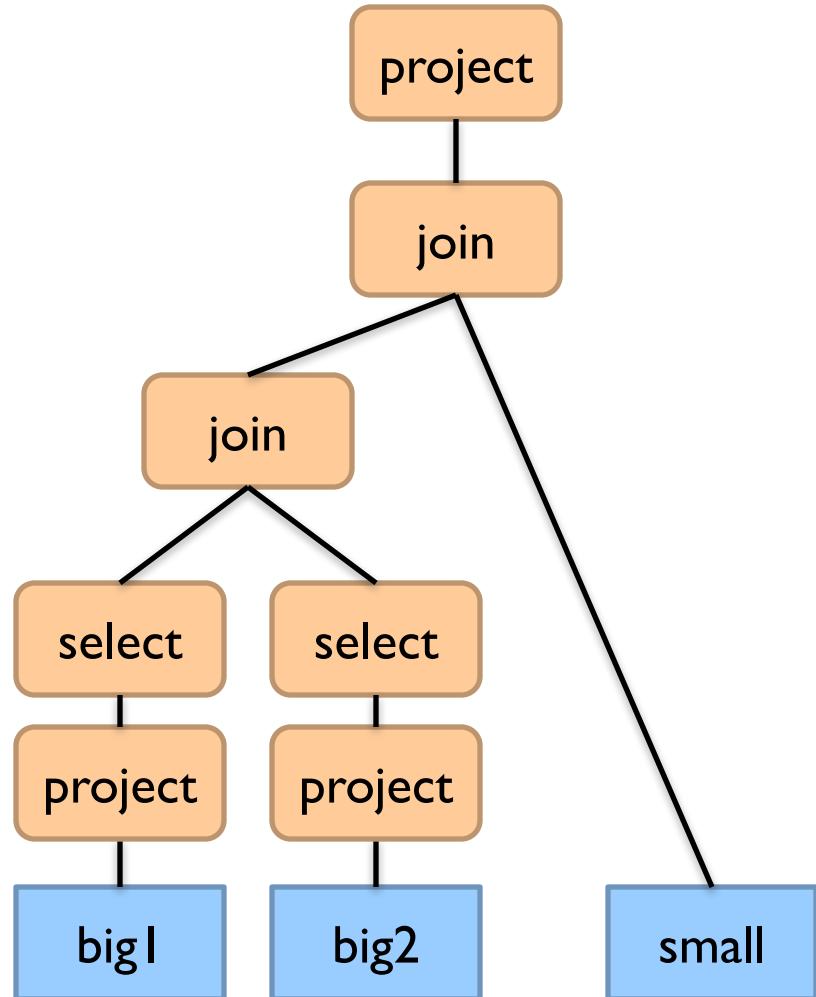
Works well with:

- Vectorized Execution
- Compiled Queries

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;
```

- Build logical plan
- Optimize logical plan
- Select physical plan



```

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

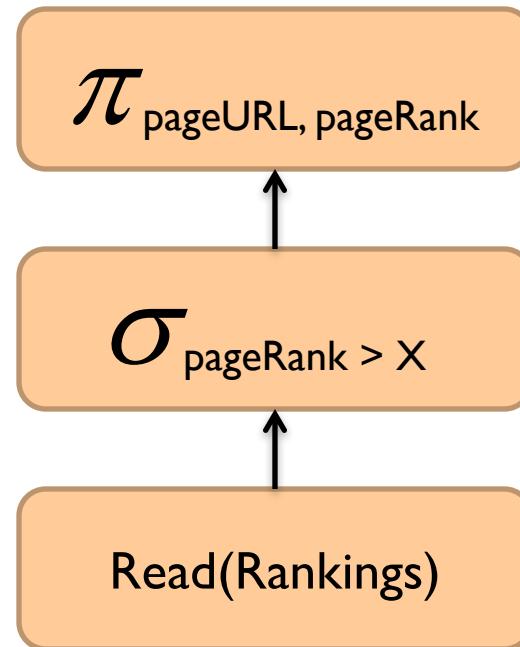
- open() Initialize, reset internal state, etc.
- next() Advance and deliver next tuple
- close() Clean up, free resources, etc.

Execution driven by repeated calls
to top of operator tree

```
open() next() next()...  
close()
```

```
open() next() next()...  
close()
```

```
open() next() next()...  
close()
```



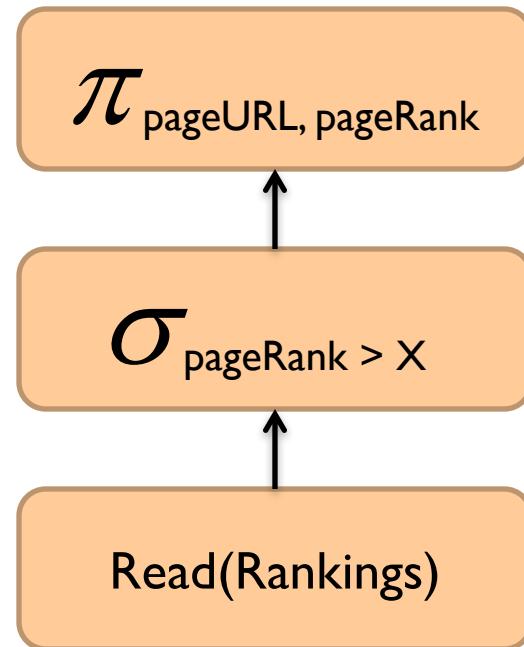
```
SELECT pageURL, pageRank  
FROM Rankings WHERE pageRank > X;
```

Very little actual computation is being done!

```
open() next() next()...  
close()
```

```
open() next() next()...  
close()
```

```
open() next() next()...  
close()
```



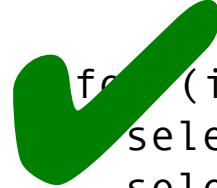
```
SELECT pageURL, pageRank  
FROM Rankings WHERE pageRank > X;
```

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

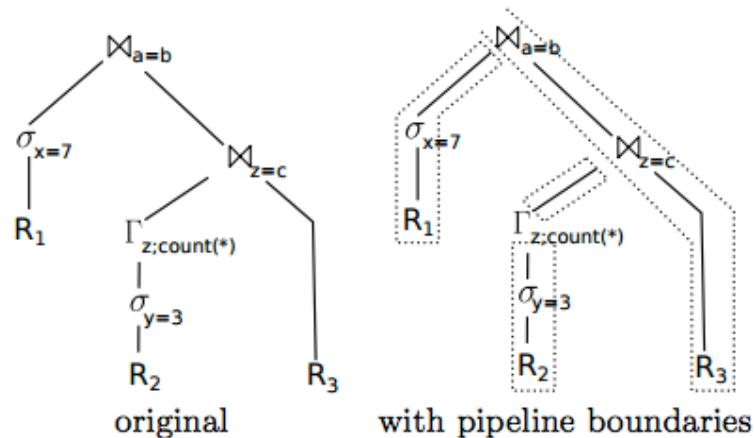
Can we do even better?

Compiled Queries

```

select      *
from        R1,R3,
           (select    R2.z,count(*)
            from      R2
            where     R2.y=3
            group by R2.z) R2
where      R1.x=7 and R1.a=R3.b and R2.z=R3.c

```



```

initialize memory of  $\bowtie_{a=b}$ ,  $\bowtie_{c=z}$ , and  $\Gamma_z$ 
for each tuple  $t$  in  $R_1$ 
  if  $t.x = 7$ 
    materialize  $t$  in hash table of  $\bowtie_{a=b}$ 
for each tuple  $t$  in  $R_2$ 
  if  $t.y = 3$ 
    aggregate  $t$  in hash table of  $\Gamma_z$ 
for each tuple  $t$  in  $\Gamma_z$ 
  materialize  $t$  in hash table of  $\bowtie_{z=c}$ 
for each tuple  $t_3$  in  $R_3$ 
  for each match  $t_2$  in  $\bowtie_{z=c}[t_3.c]$ 
    for each match  $t_1$  in  $\bowtie_{a=b}[t_3.b]$ 
      output  $t_1 \circ t_2 \circ t_3$ 

```

Compiled Queries

Example LLVM query template

```
define internal void @scanConsumer(%8* %executionState, %Fragment_R2* %data) {
body:
...
%columnPtr = getelementptr inbounds %Fragment_R2* %data, i32 0, i32 0
%column = load i32** %columnPtr, align 8
%columnPtr2 = getelementptr inbounds %Fragment_R2* %data, i32 0, i32 1
%column2 = load i32** %columnPtr2, align 8
... (loop over tuples, currently at %id, contains label %cont17)
%yPtr = getelementptr i32* %column, i64 %id
%y = load i32* %yPtr, align 4
%cond = icmp eq i32 %y, 3
br i1 %cond, label %then, label %cont17
then:
%zPtr = getelementptr i32* %column2, i64 %id
%z = load i32* %zPtr, align 4
%hash = urem i32 %z, %hashTableSize
%hashSlot = getelementptr %"HashGroupify::Entry"** %hashTable, i32 %hash
%hashIter = load %"HashGroupify::Entry"** %hashSlot, align 8
%cond2 = icmp eq %"HashGroupify::Entry" * %hashIter, null
br i1 %cond, label %loop20, label %else26
... (check if the group already exists, starts with label %loop20)
else26:
%cond3 = icmp le i32 %spaceRemaining, i32 8
br i1 %cond, label %then28, label %else47
... (create a new group, starts with label %then28)
else47:
%ptr = call i8* @_ZN12HashGroupify15storeInputTupleEmj
    (%"HashGroupify" * %1, i32 hash, i32 8)
... (more loop logic)
}
```

1. locate tuples in memory
2. loop over all tuples
3. filter $y = 3$
4. hash z
5. lookup in hash table (C++ data structure)
6. not found, check space
7. full, call C++ to allocate mem or spill

Advantages of Column Stores

Inherent advantages:

- Better compression
- Read efficiency

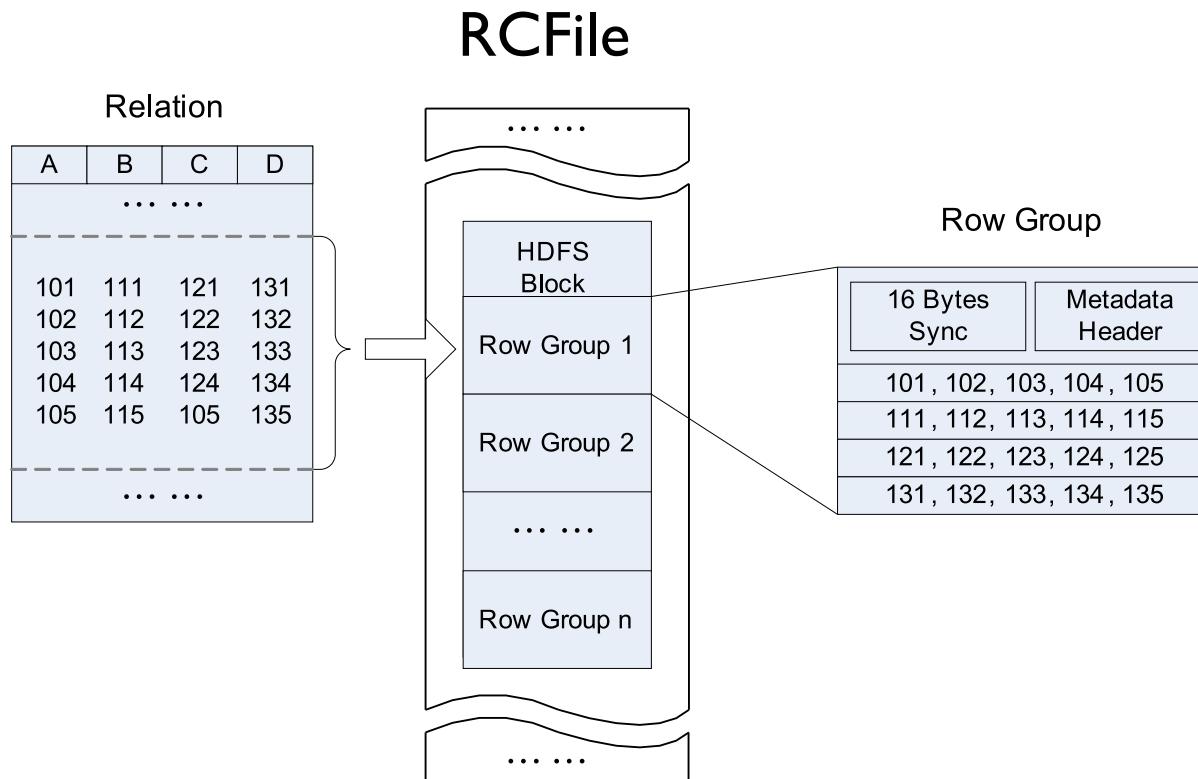
Works well with:

- Vectorized Execution
- Compiled Queries

These are well-known in traditional databases...

Why not in Hadoop?

Why not in Hadoop? No reason why not!





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



Compiled Queries?



```
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!

Advantages of Column Stores

Inherent advantages:

- Better compression
- Read efficiency

Works well with:

- Vectorized Execution
- Compiled Queries

Hadoop can adopt all of these optimizations!

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



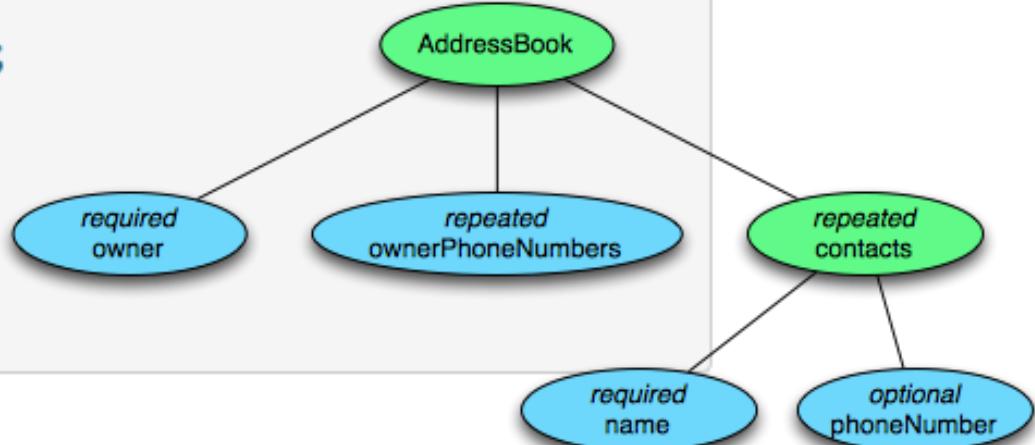
Optional and Repeated Elements

Schema: List of Strings	Data: ["a", "b", "c", ...]
<pre>message ExampleList { repeated string list; }</pre>	<pre>{ list: "a", list: "b", list: "c", ... }</pre>

Schema: Map of strings to strings	Data: { "AL" => "Alabama", ... }
<pre>message ExampleMap { repeated group map { required string key; optional string value; } }</pre>	<pre>{ map: { key: "AL", value: "Alabama" }, map: { key: "AK", value: "Alaska" }, ... }</pre>

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

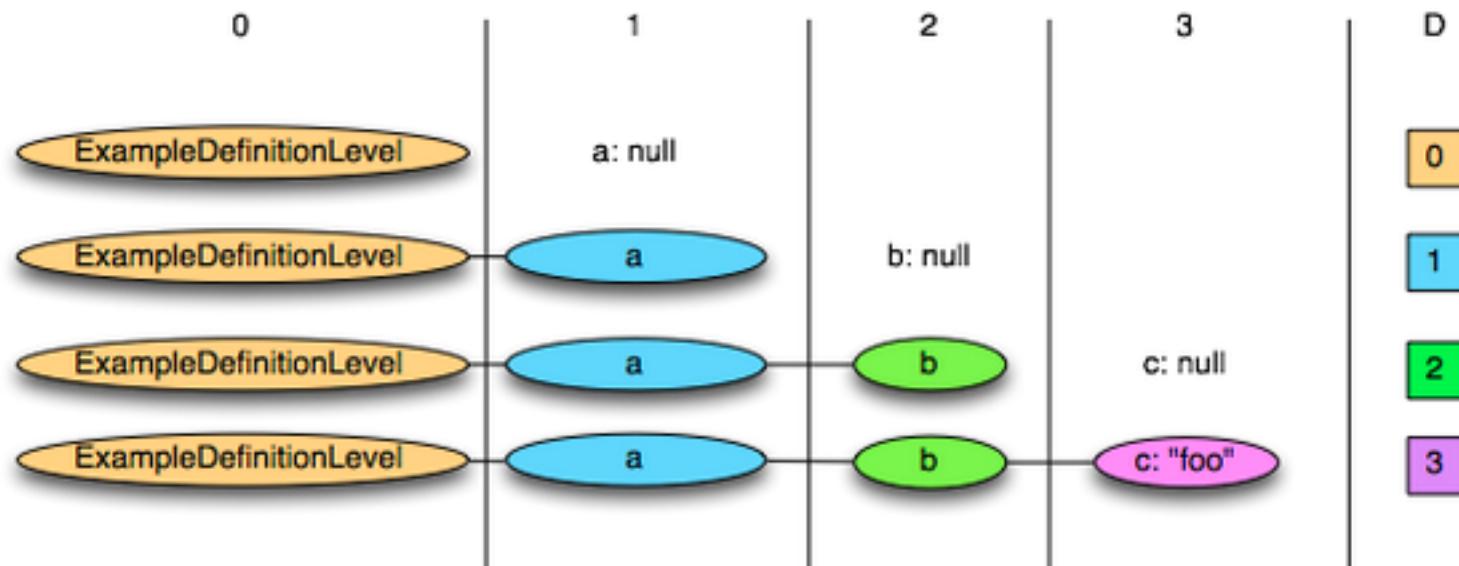
What other information
do we need to store?

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)

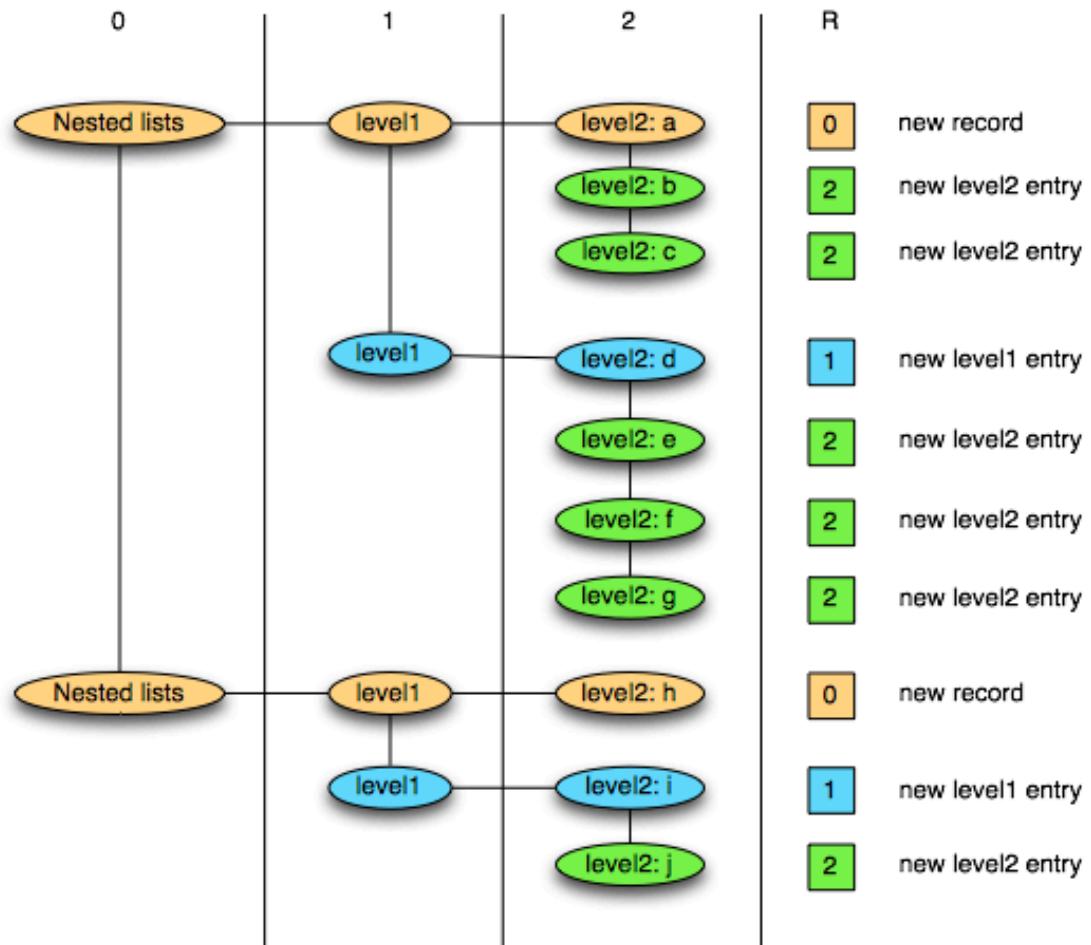
Definition Level: Illustration



Repetition Level

Schema:	Data:
<pre>message nestedLists { repeated group level1 { repeated string level2; } }</pre>	<pre>{ level1: { level2: a level2: b level2: c }, level1: { level2: d level2: e level2: f level2: g } } { level1: { level2: h }, level1: { level2: i level2: j } }</pre>

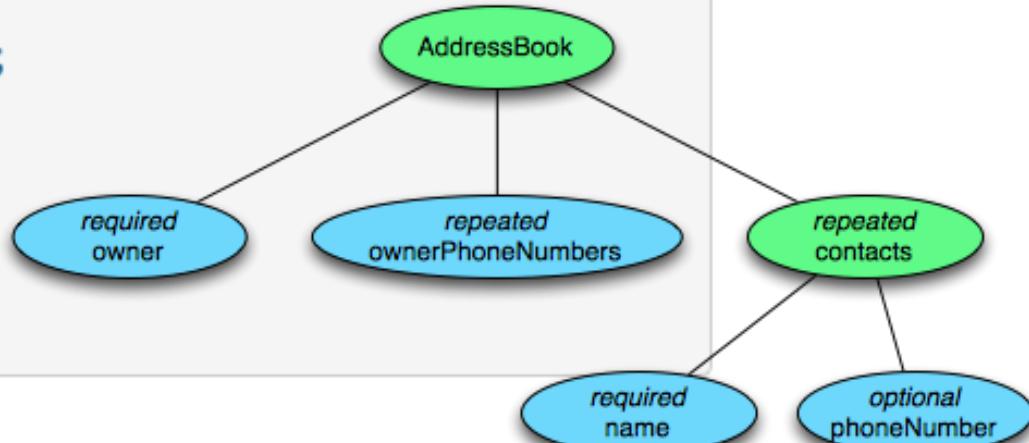
Repetition Level: Illustration



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 <i>required</i>)	0 (no repetition)
ownerPhoneNumbers	1	1 (<i>repeated</i>)
contacts.name	1 (name is <i>required</i>)	1 (<i>contacts</i> is <i>repeated</i>)
contacts.phoneNumber	2 (phoneNumber is <i>optional</i>)	1 (<i>contacts</i> is <i>repeated</i>)

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. Anonymous"  
}
```

Project onto contacts.phoneNumber

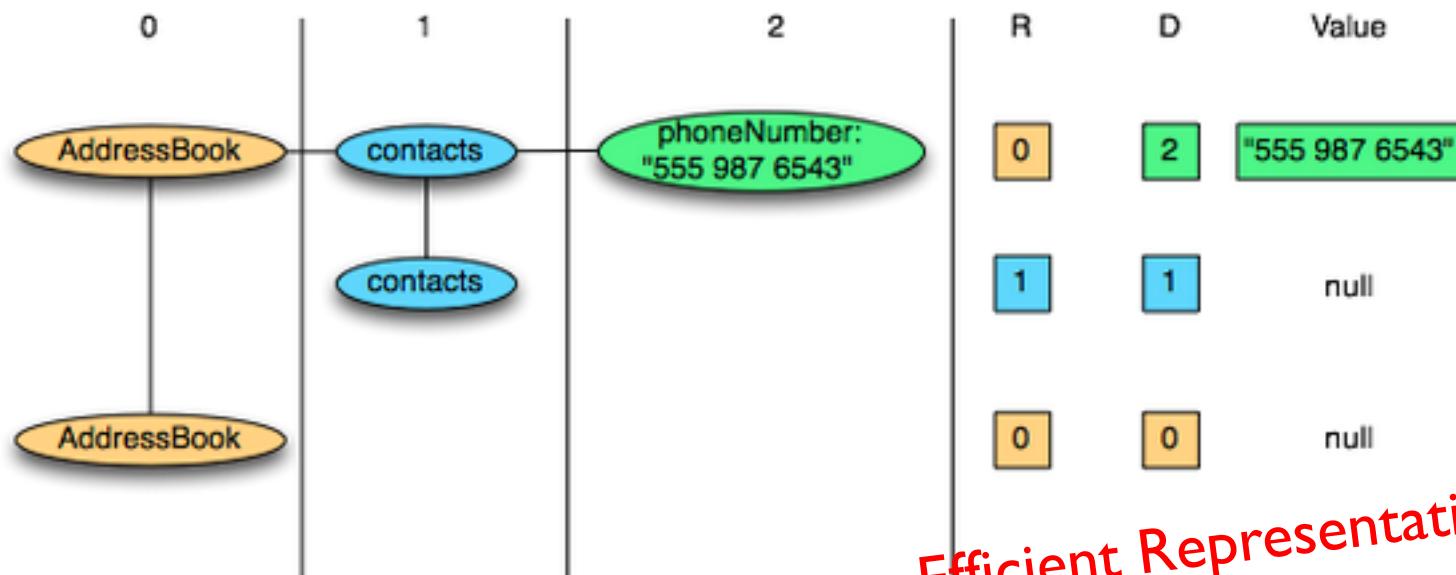


```
AddressBook {  
    contacts: {  
        phoneNumber: "555 987 6543"  
    }  
    contacts: {}  
}  
AddressBook {  
}
```

Physical Layout

Columnar Decomposition

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



Key Ideas

Binary representations are good

Binary representations need schemas

Schemas allow logical/physical separation

Logical/physical separation allows you to do cool things

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



Indexes are a good thing!

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 chuck's out most of the data *Stupid.*



“Trying to find a needle in a haystack... with a snowplow”
@squarecog

Hadoop + Full-Text Indexes

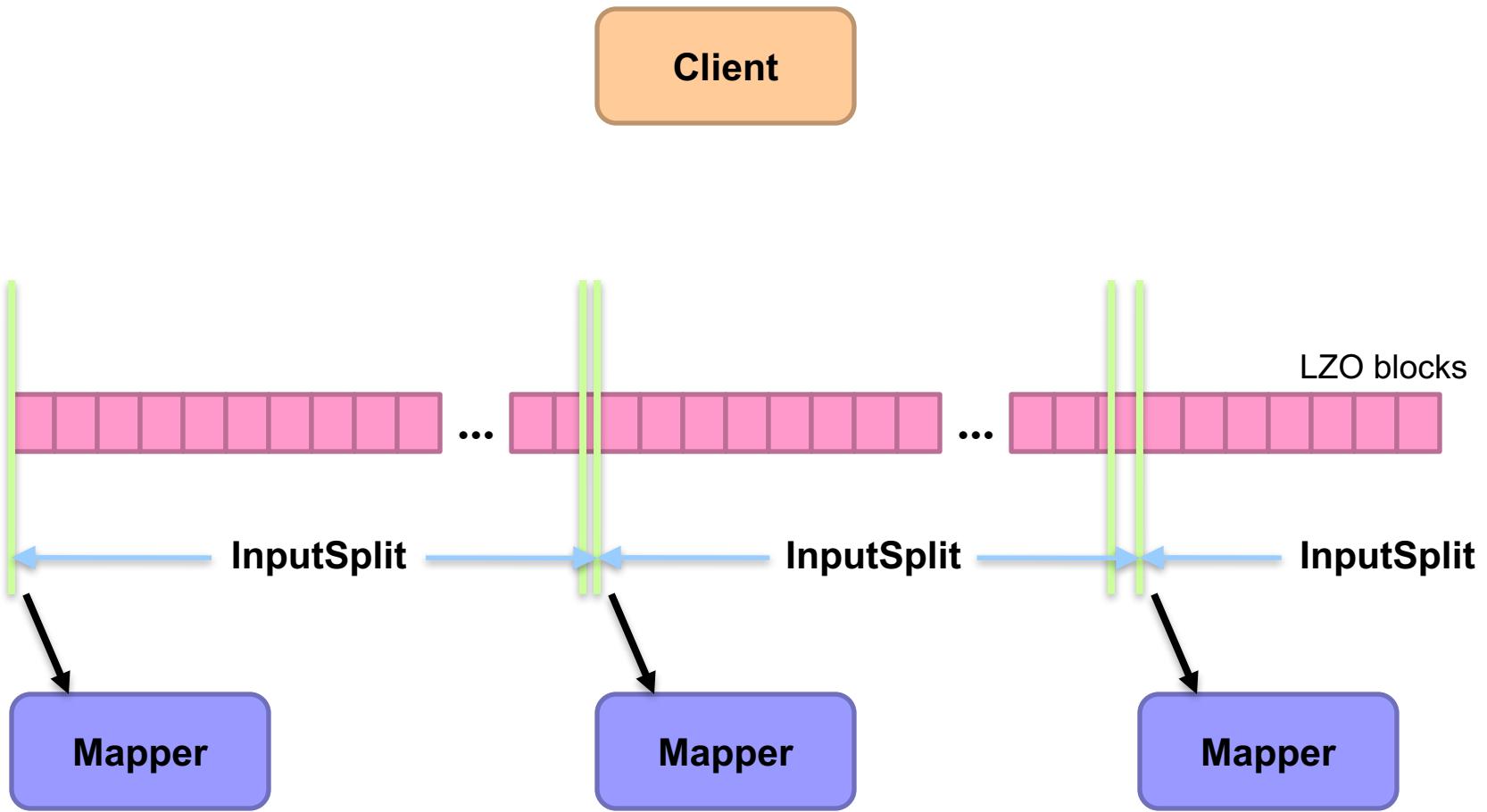
```
status = load '/tables/statuses/2011/03/01'  
    using StatusProtobufPigLoader()  
    as (id: long, user_id: long, text: chararray, ...);  
  
filtered = filter status by text matches '.*\\bhadoop\\b.*';  
...
```

Pig performs a brute force scan

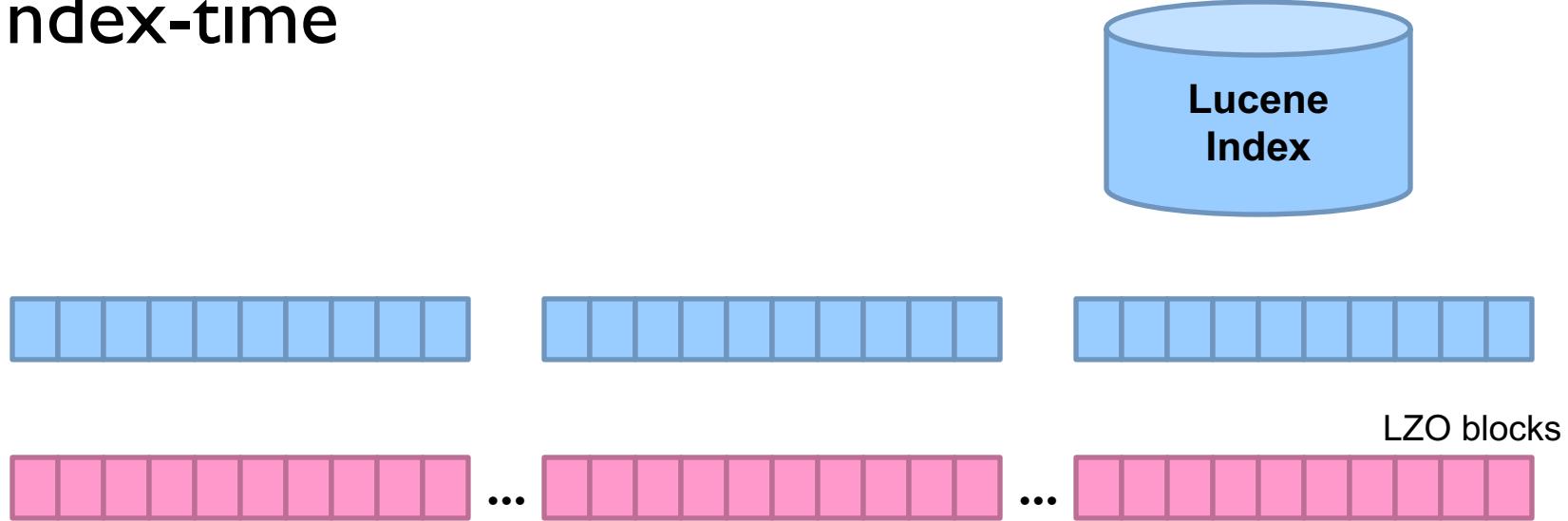
Then promptly chuck's out most of the data *Stupid.*

Uhh... how about an index?

Use Lucene full-text index



Index-time

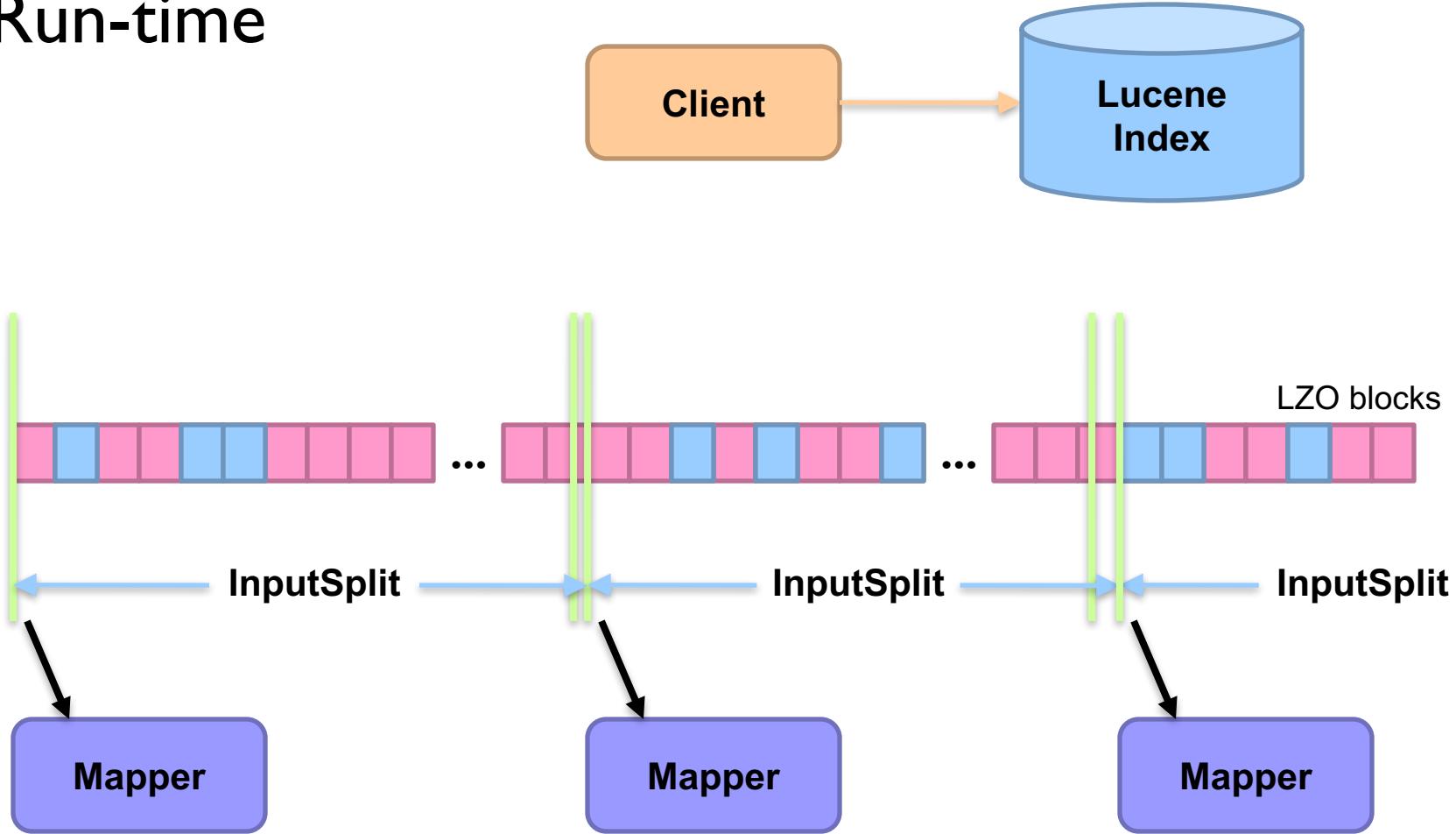


Index for selection on tweet content

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

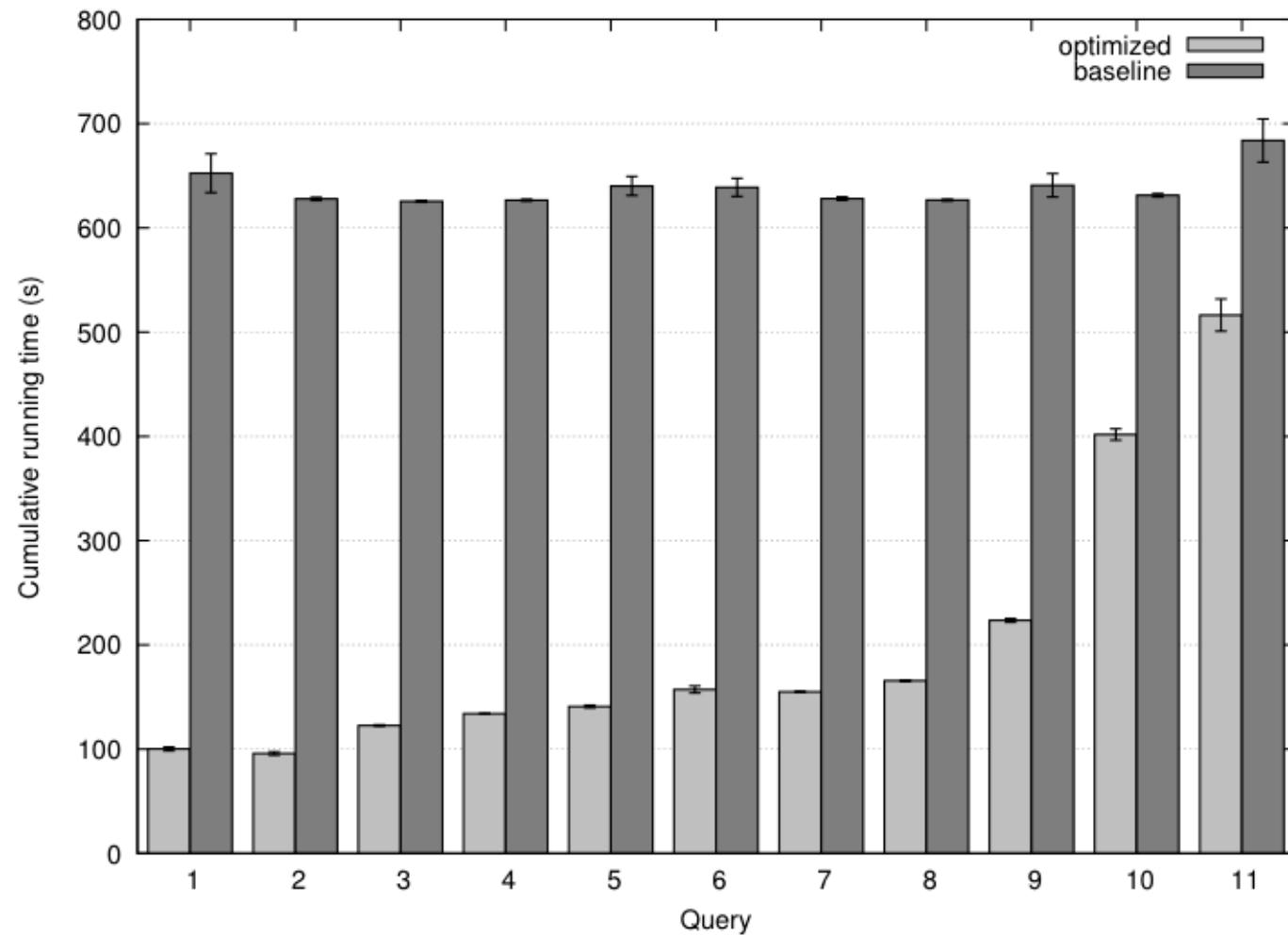
Experiments

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}



Analytical model

Task: prediction LZO blocks scanned by selectivity

Poisson model: $P(\text{observing } k \text{ occurrences in a block})$

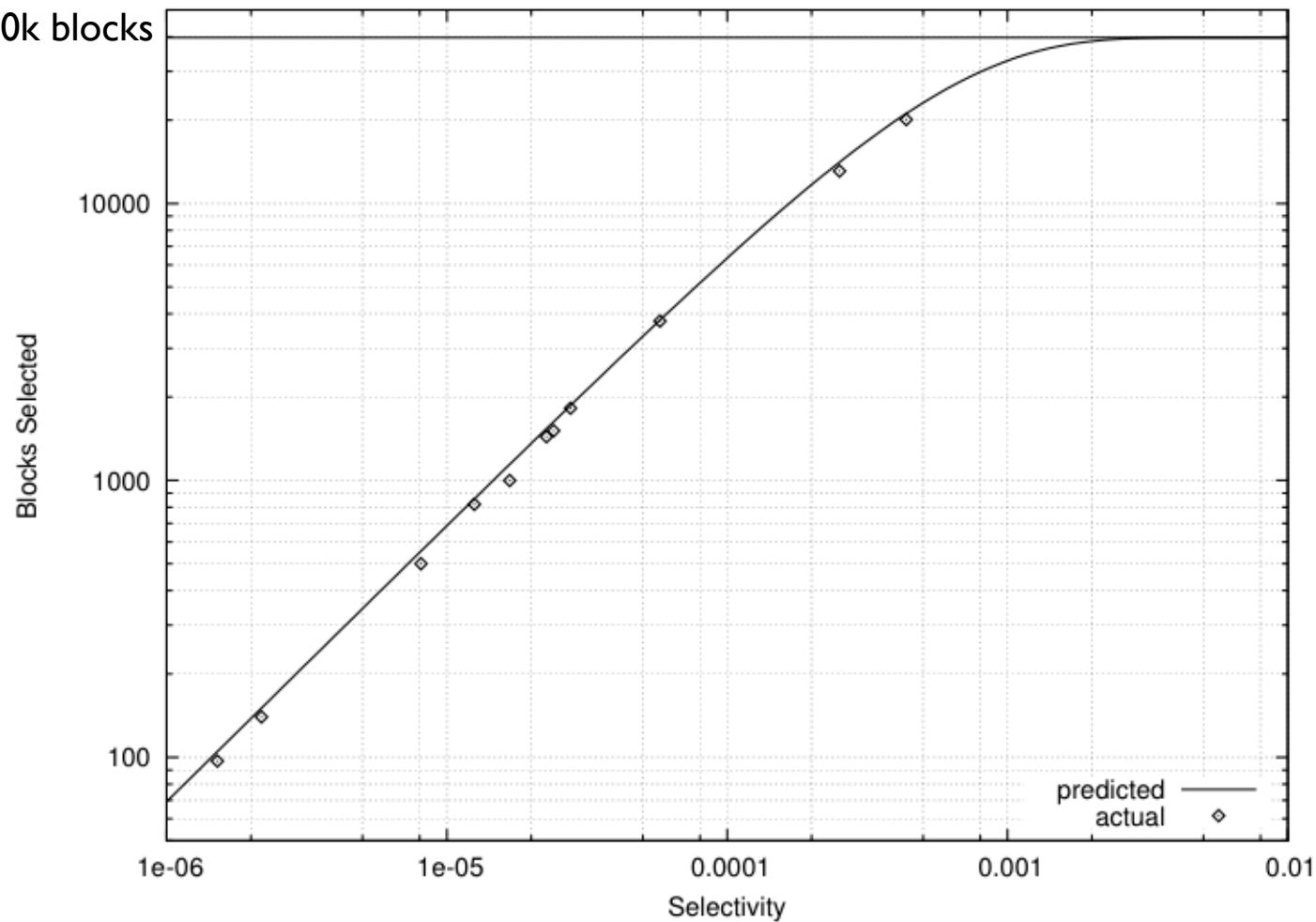
$$f(k; \lambda) = \frac{\lambda^k e^{-\lambda}}{k!}$$

$E(\text{fraction of blocks scanned})$

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

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

Total: ~40k blocks



But: can predict *a priori*!

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

A photograph of a traditional Japanese rock garden. In the foreground, a gravel path is raked into fine, parallel lines. Several large, dark, irregular stones are scattered across the garden. A small, shallow pond is visible in the middle ground, surrounded by more stones and some low-lying green plants. In the background, there are more stones, some small trees, and the wooden buildings of a residence with tiled roofs.

Questions?