Data-Intensive Distributed Computing

CS 431/631 451/651 (Fall 2019)

Part 5: Analyzing Relational Data (3/3)

October 24, 2019

Ali Abedi

These slides are available at https://www.student.cs.uwaterloo.ca/~cs451

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

Source: Blog post by DeWitt and Stonebraker

Hadoop vs. Databases: Grep

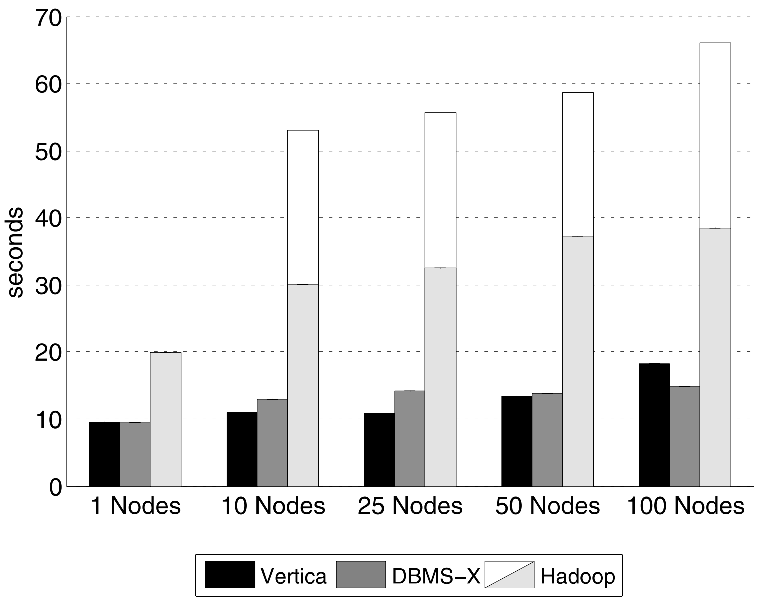


Figure 4: Grep Task Results – 535MB/node Data Set

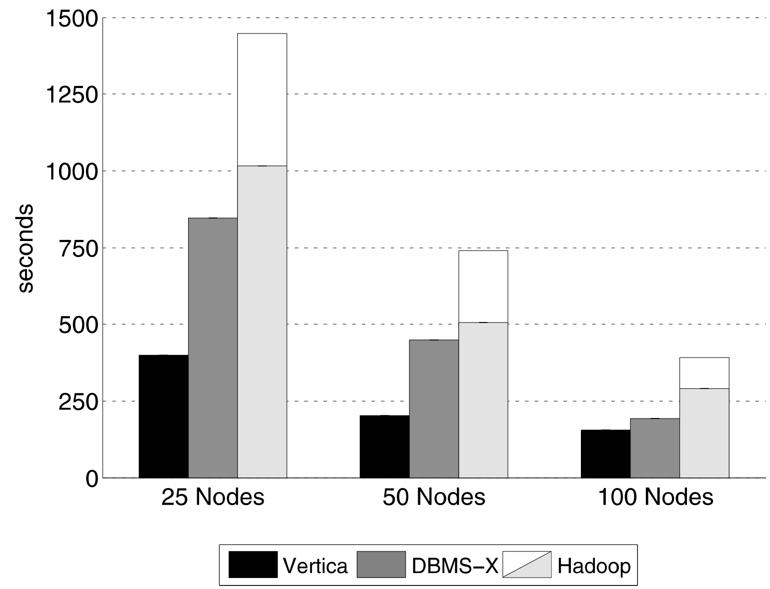


Figure 5: Grep Task Results – 1TB/cluster Data Set

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

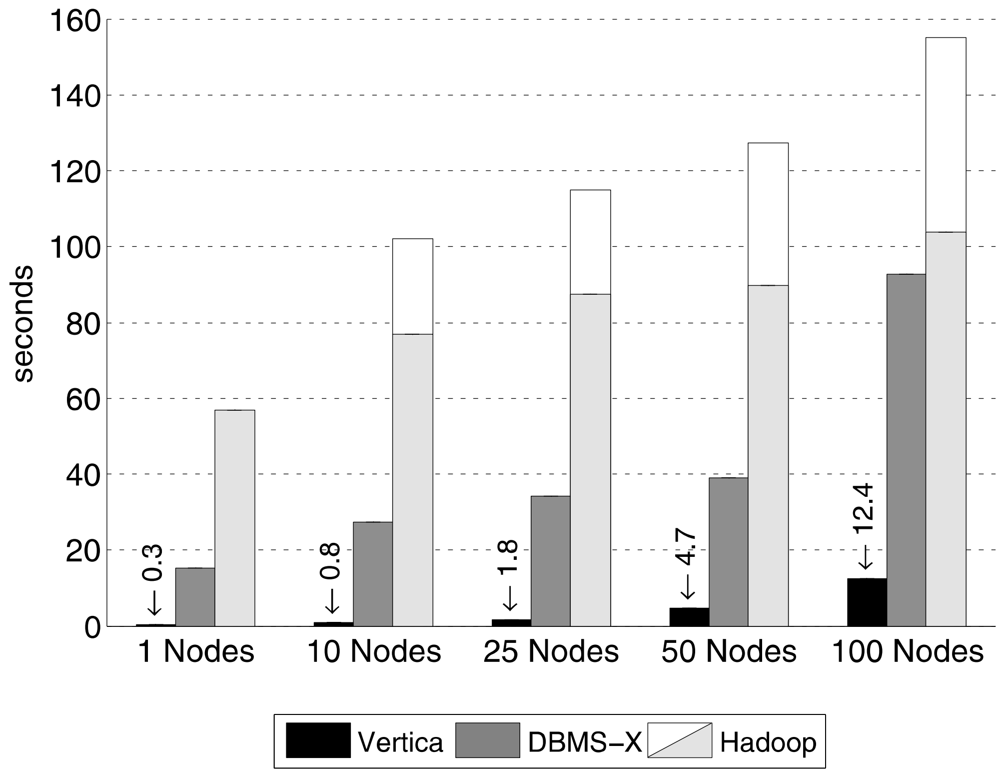


Figure 6: Selection Task Results

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

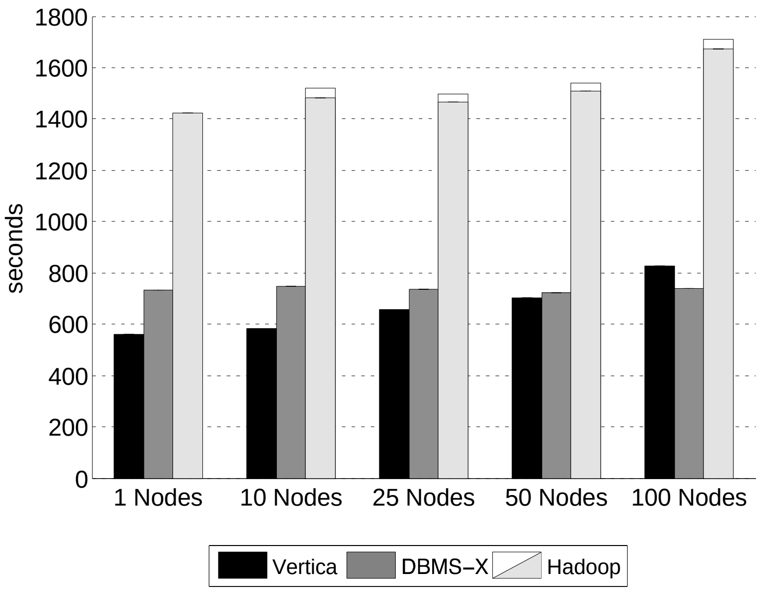


Figure 7: Aggregation Task Results (2.5 million Groups)

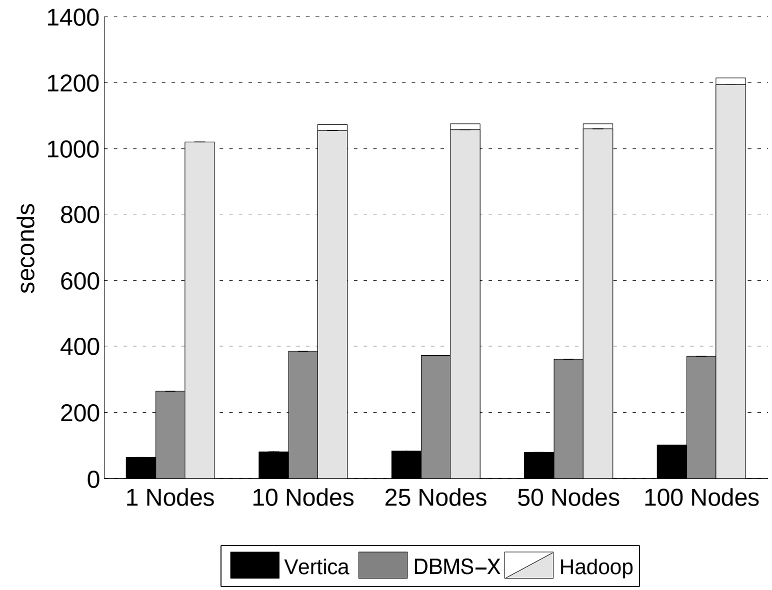


Figure 8: Aggregation Task Results (2,000 Groups)

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

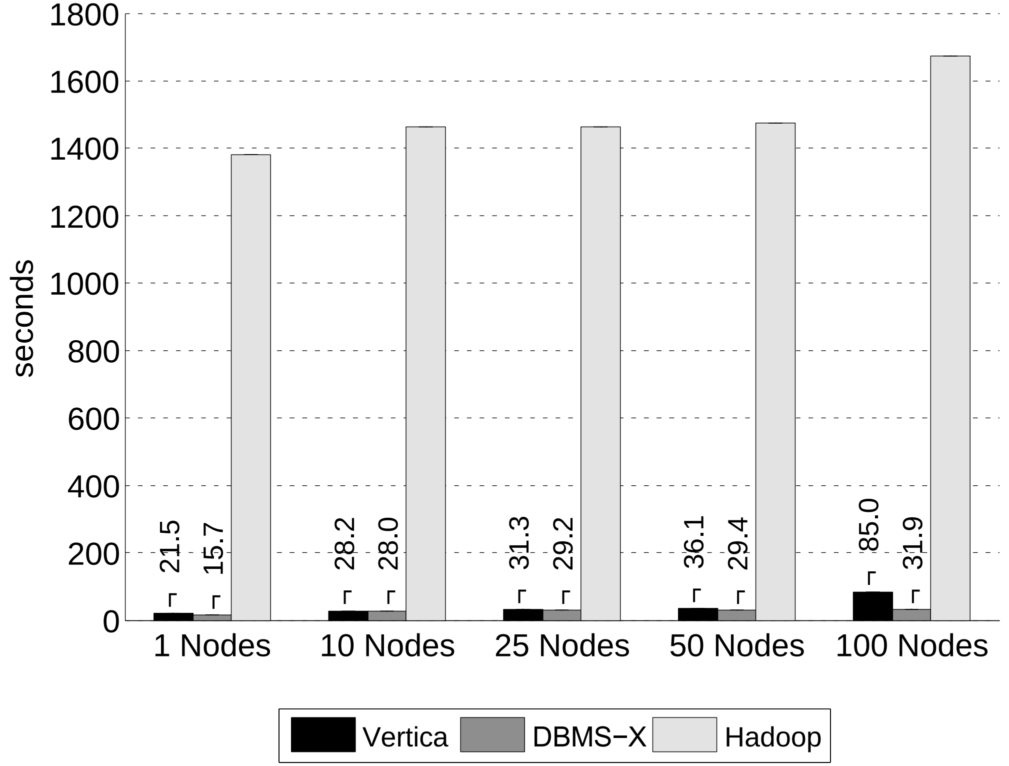


Figure 9: Join Task Results

```
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.

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

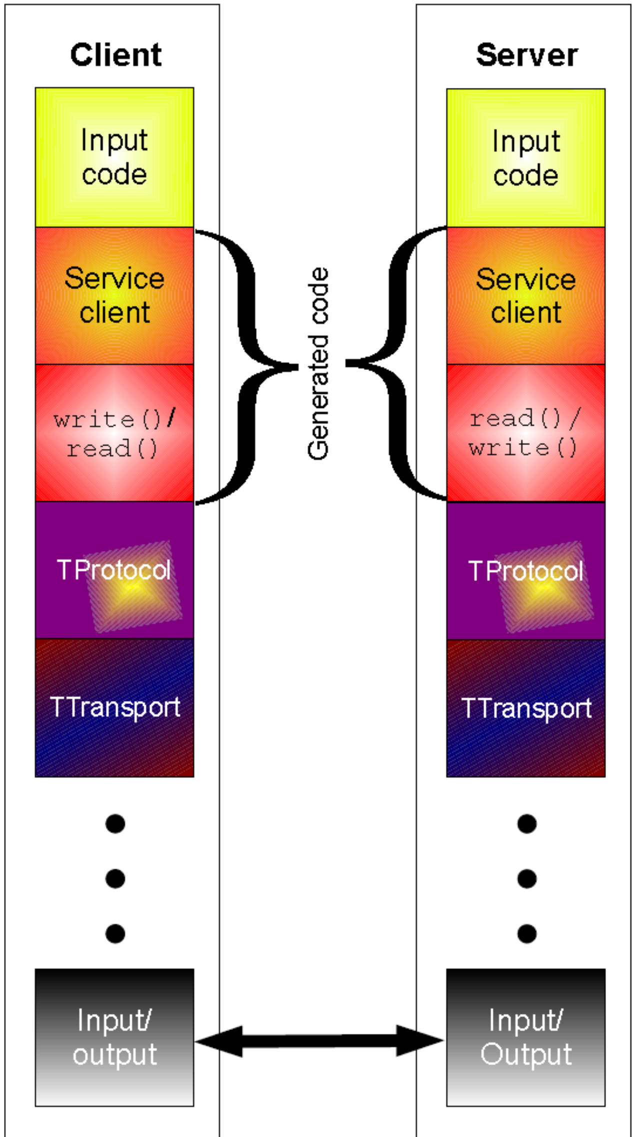
Provides a DDL with numerous language bindings

- 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

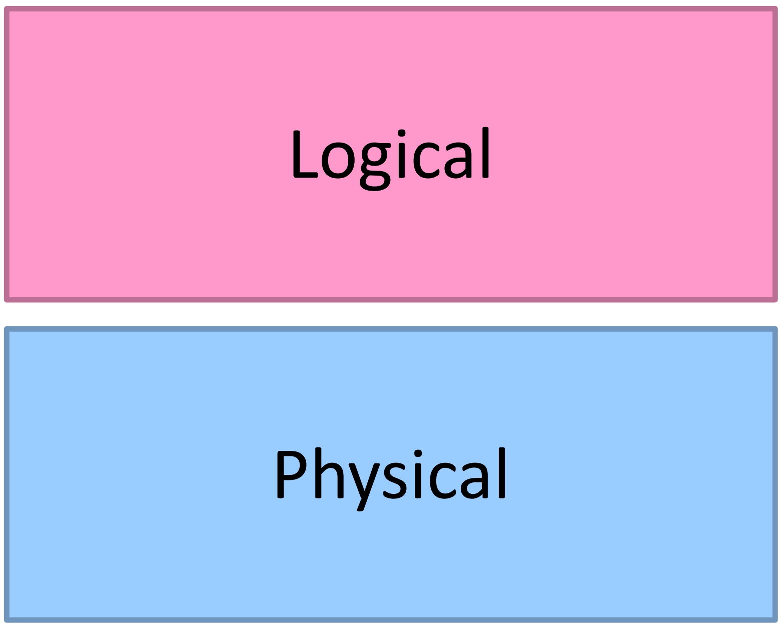


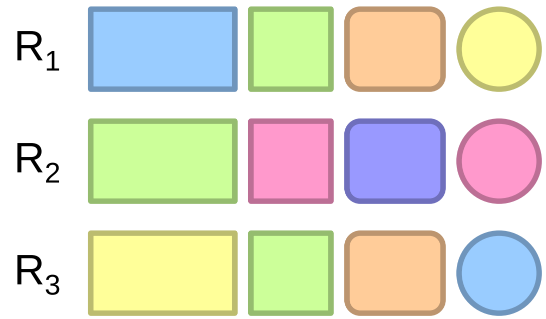

```
struct Tweet {
 1: required i32 userId;
 2: required string userName;
 3: required string text;
 4: optional Location loc;
}

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

Why not…

XML or JSON?

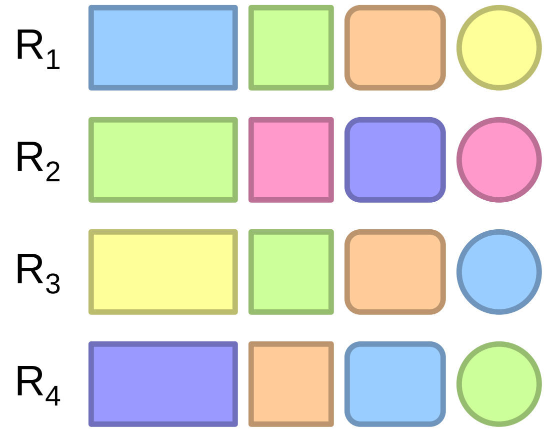
REST?



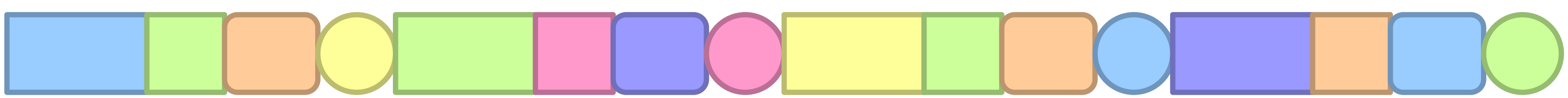


How bytes are actually represented in storage…

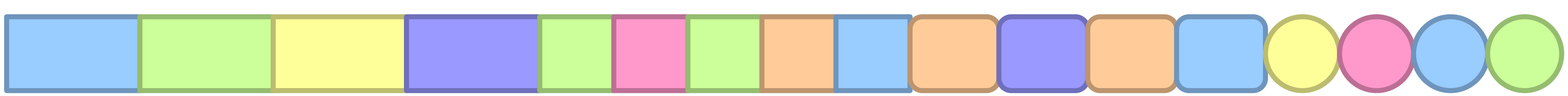
Row vs. Column Stores



Row store



Column store



Row vs. Column Stores

Row stores

Easier to modify a record: in-place updates

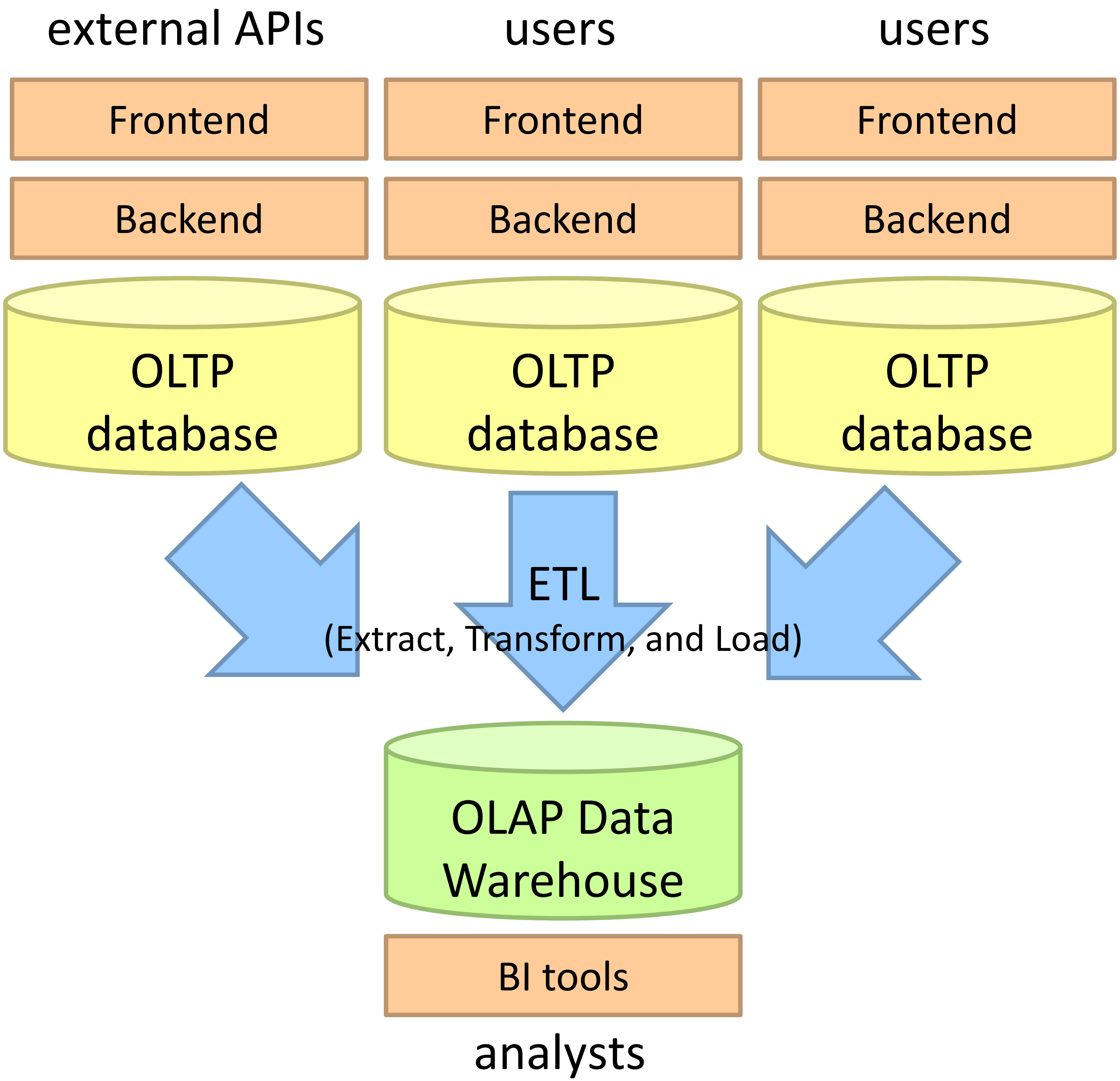
Might read unnecessary data when processing

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

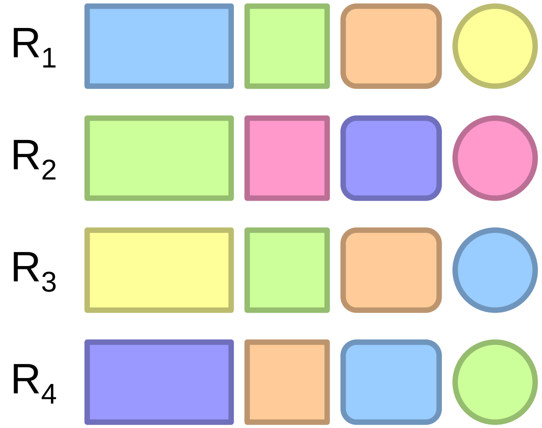
Works well with:

Vectorized Execution

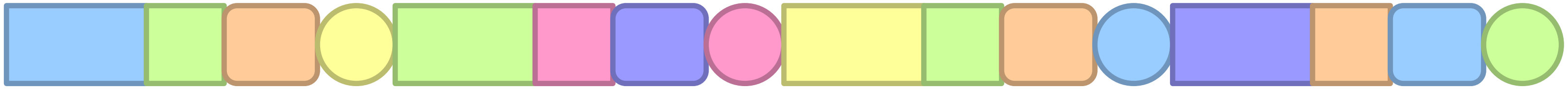
Compiled Queries

These are well-known in traditional databases…

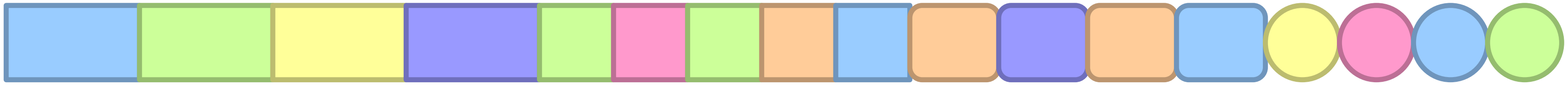
Row vs. Column Stores: Compression



Row store



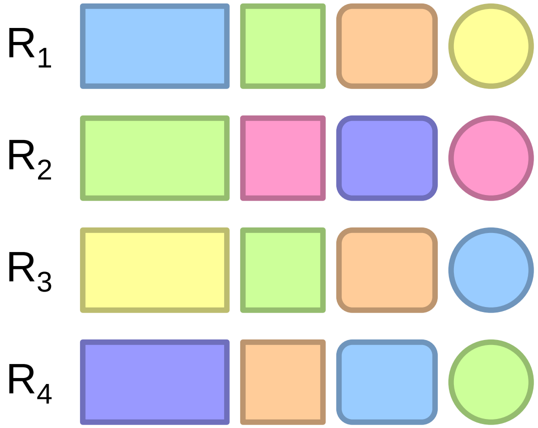
Column store



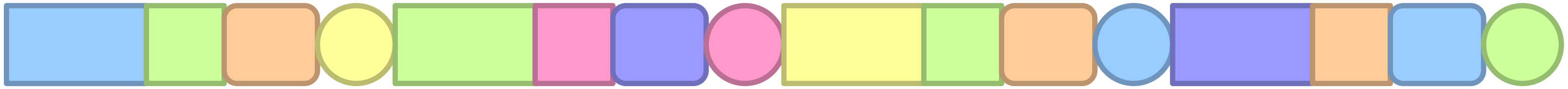
This compresses better with off-the-shelf tools, e.g., gzip.

Why?

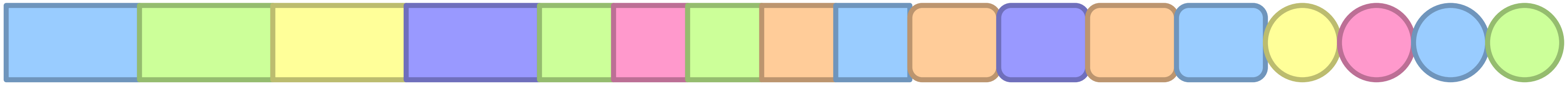
Row vs. Column Stores: Compression



Row store



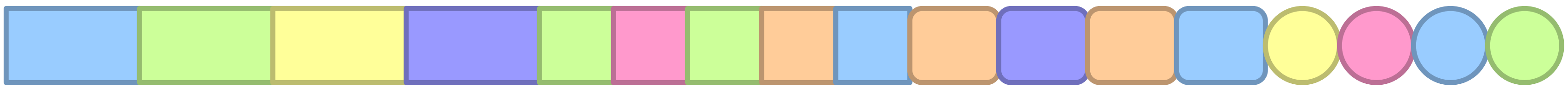
Column store



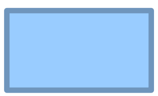
Additional opportunities for smarter compression…

Columns Stores: RLE

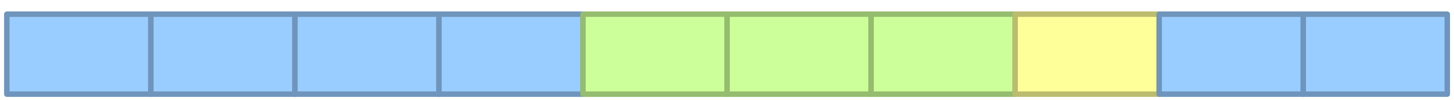
Column store



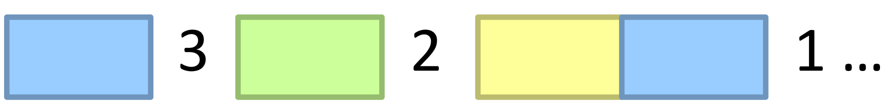
Run-length encoding example:

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

In reality:

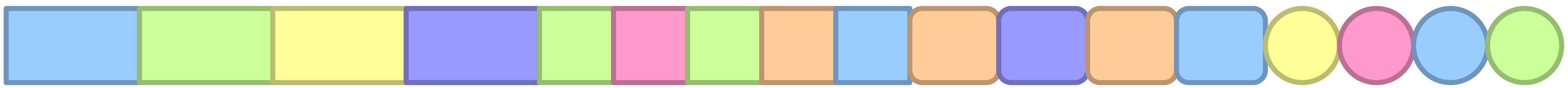
 …

Encode:



Columns Stores: Integer Coding

Column store



Say you're coding a bunch of integers…

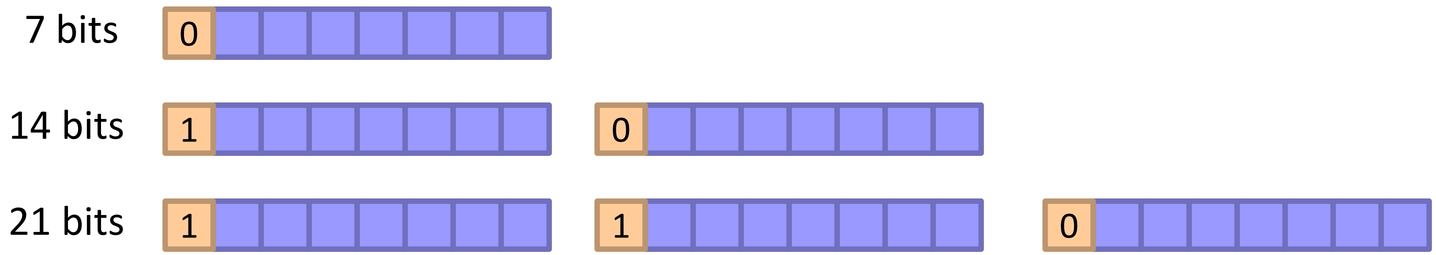
Remember this? (Part 3)

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



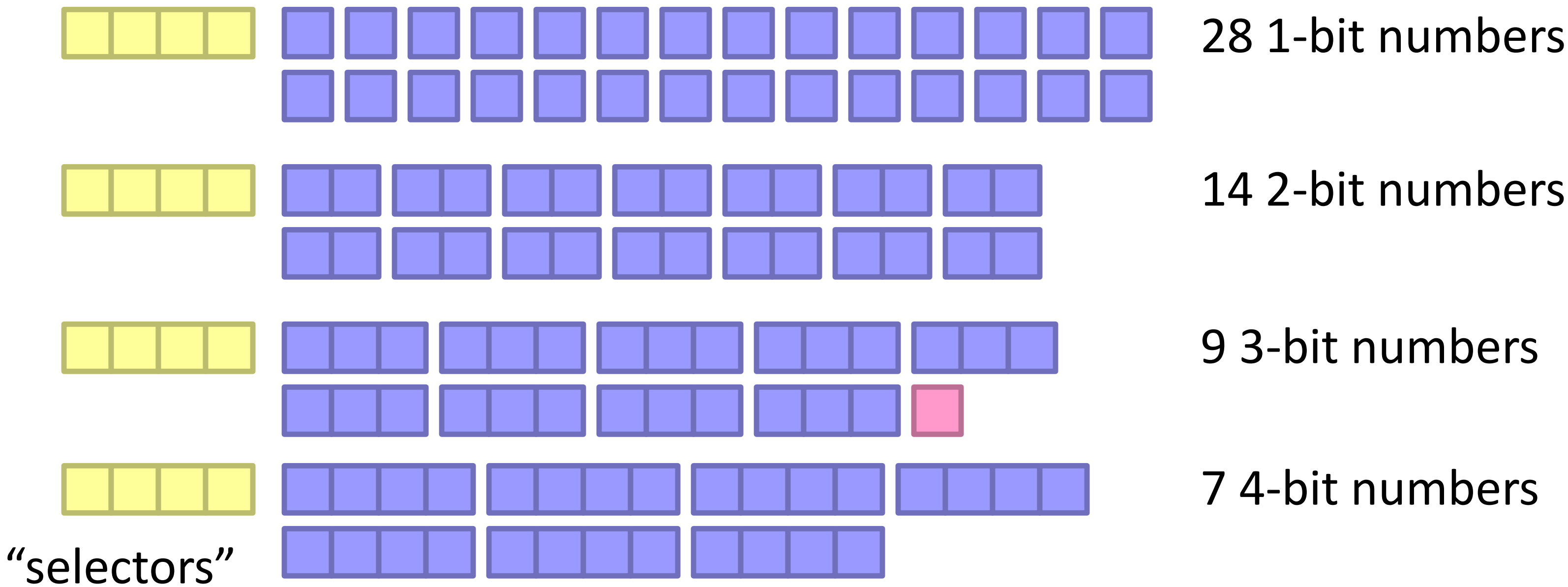
Works okay, easy to implement…

Beware of branch mispredicts!

Remember this?
(Part 3)

Simple-9

How many different ways can we divide up 28 bits?



(9 total ways)

Efficient decompression with hard-coded decoders

Simple Family – general idea applies to 64-bit words, etc.

Beware of branch mispredicts?

Advantages of Column Stores

Inherent advantages:

Better compression

Read efficiency

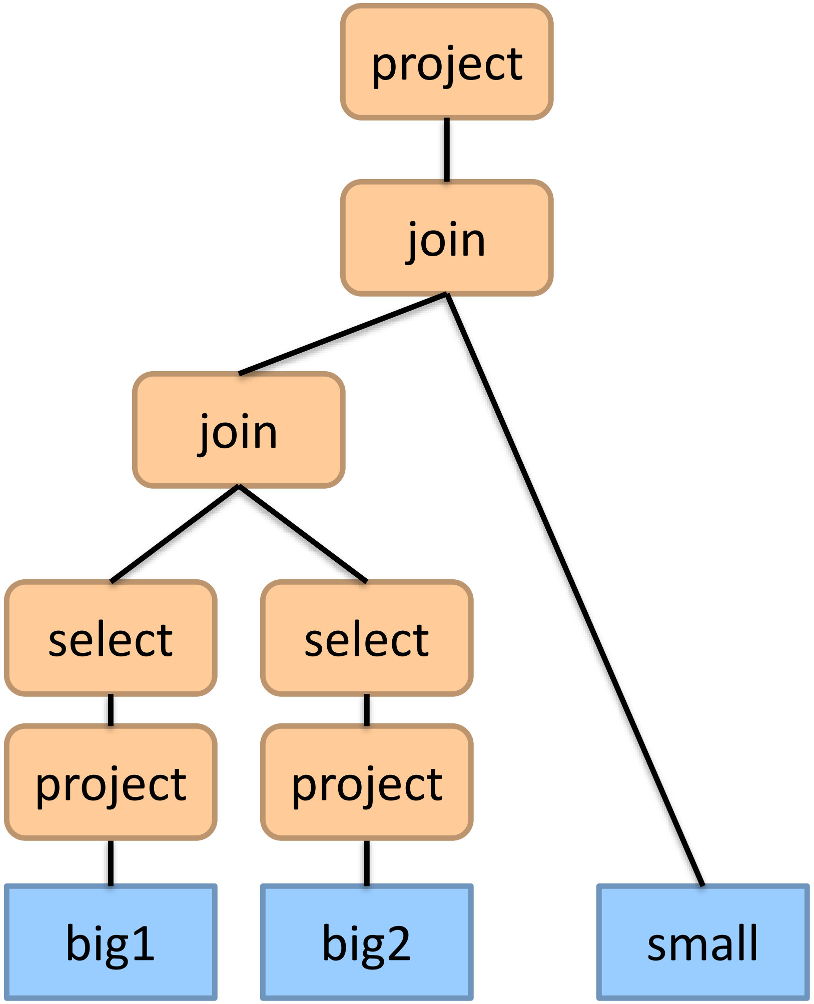
Works well with:

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

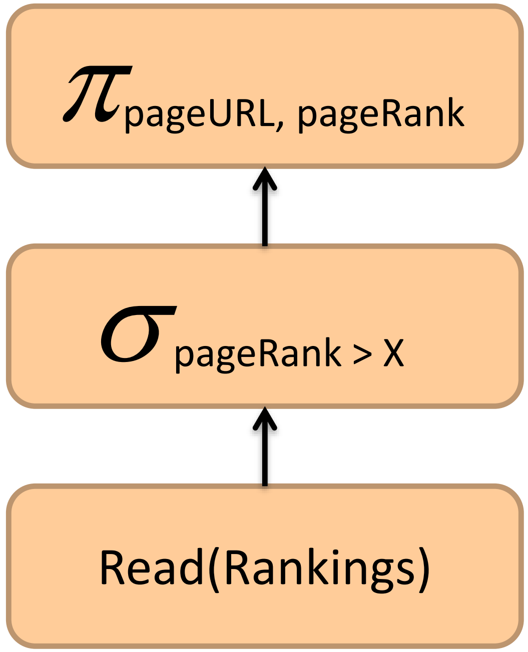
open()	Initialize, reset internal state, etc.
next()	Advance and deliver next tuple
close()	Clean up, free resources, etc.

Execution driven by repeated calls to top of operator tree

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

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

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



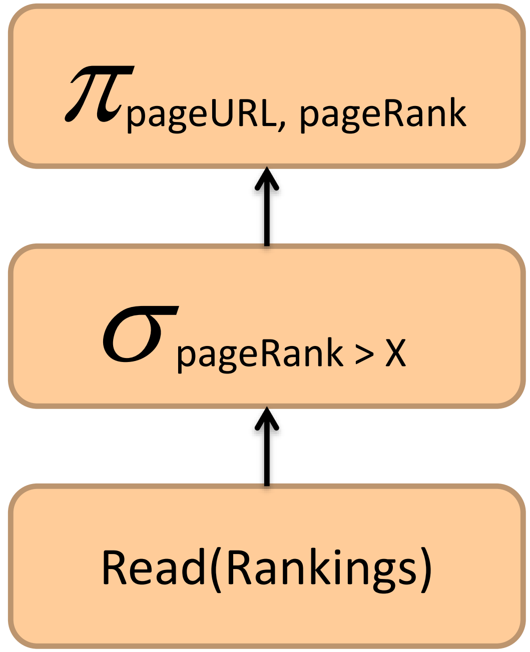
SELECT pageURL, pageRank
FROM Rankings WHERE pageRank > X;

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

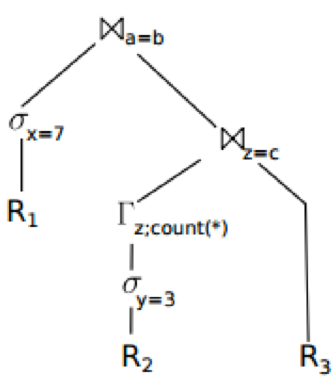
next() returns a vector of tuples

All operators rewritten to work on vectors of tuples

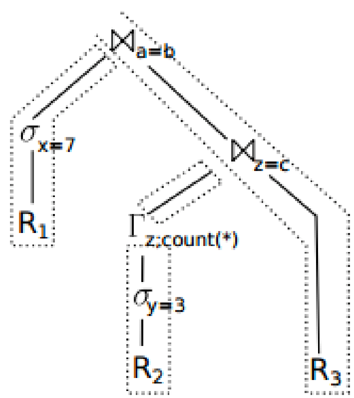
Can we do even better?

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
```



original



with pipeline boundaries

initialize memory of $\bowtie_{a=b}$, $\bowtie_{c=z}$, and Γ_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 Γ_z
for each tuple t in Γ_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 $t_1 \circ t_2 \circ t_3$

Source: Neumann (2011) Efficiently Compiling Efficient Query Plans for Modern Hardware. VLDB.

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

Source: Neumann (2011) Efficiently Compiling Efficient Query Plans for Modern Hardware. VLDB.

Advantages of Column Stores

Inherent advantages:

Better compression

Read efficiency

Works well with:

Vectorized Execution

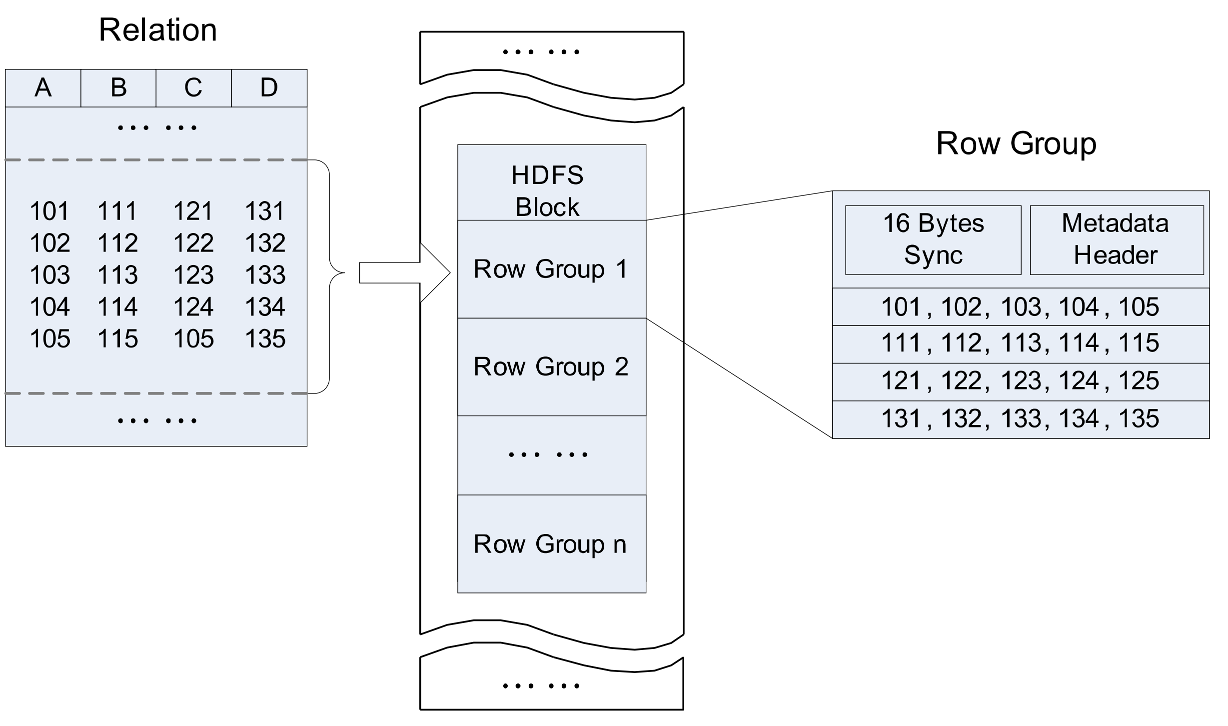
Compiled Queries

These are well-known in traditional databases…

Why not in Hadoop?

Why not in Hadoop?
No reason why not!

RCFile



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

✓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!

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

Source: Blog post by DeWitt and Stonebraker