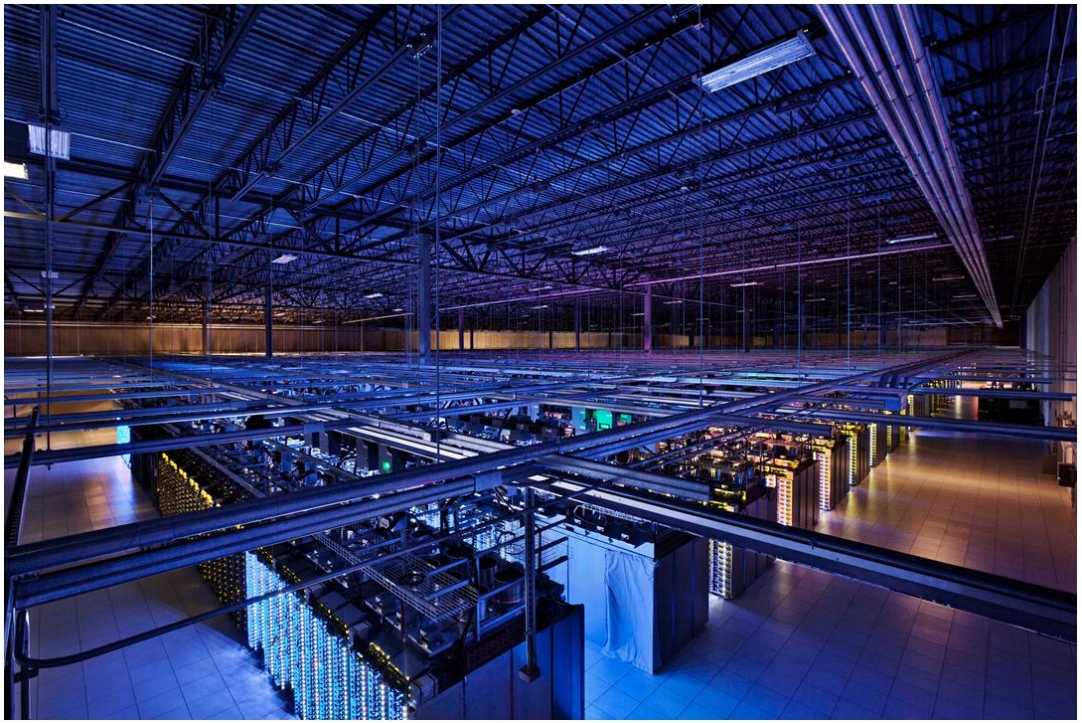
CWI
Centrum Wiskunde & Informatica

Large Scale Data Engineering

SQL on Big Data



THE DEBATE: DATABASE SYSTEMS VS MAPREDUCE

A major step backwards?

Michael Stonebraker
Turing Award Winner 2015

- 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

Known and unknown unknowns

- Databases only help if you know what questions to ask
  - "Known unknowns"
- What's if you don't know what you're looking for?
  - "Unknown unknowns"

ETL: redux

- Often, with noisy datasets, ETL *is* the analysis!
- Note that ETL necessarily involves brute force data scans
- L, then E and T?

Relational databases vs. MapReduce

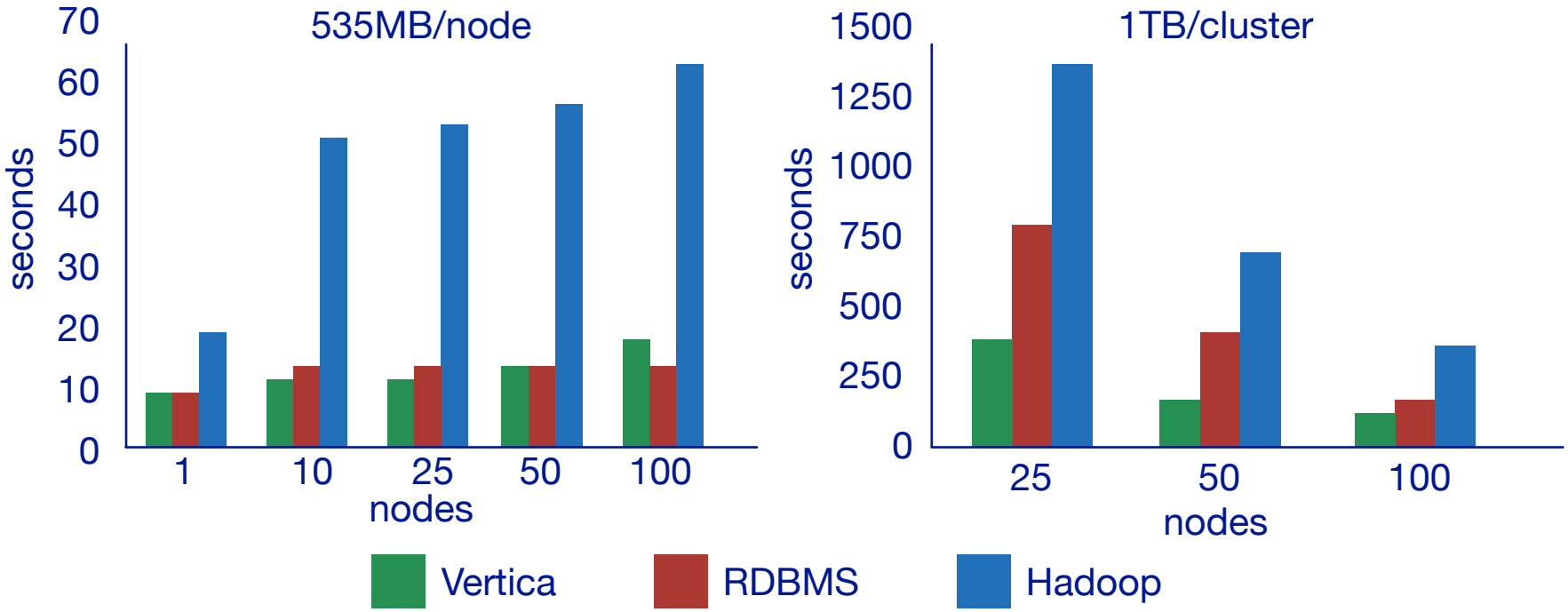
- Relational databases:
  - Multipurpose: analysis and transactions; batch and interactive
  - Data integrity via ACID transactions
  - Lots of tools in software ecosystem (for ingesting, reporting, etc.)
  - Supports SQL (and SQL integration, e.g., JDBC)
  - Automatic SQL query optimization
- MapReduce (Hadoop):
  - Designed for large clusters, fault tolerant
  - Data is accessed in “native format”
  - Supports many query languages
  - Programmers retain control over performance
  - Open source

Source: O’Reilly Blog post by Joseph Hellerstein (11/19/2008)

Philosophical differences

- Parallel relational databases
  - Schema on write
  - Failures are relatively infrequent
  - “Possessive” of data
  - Mostly proprietary
- MapReduce
  - Schema on read
  - Failures are relatively common
  - In situ data processing
  - Open source

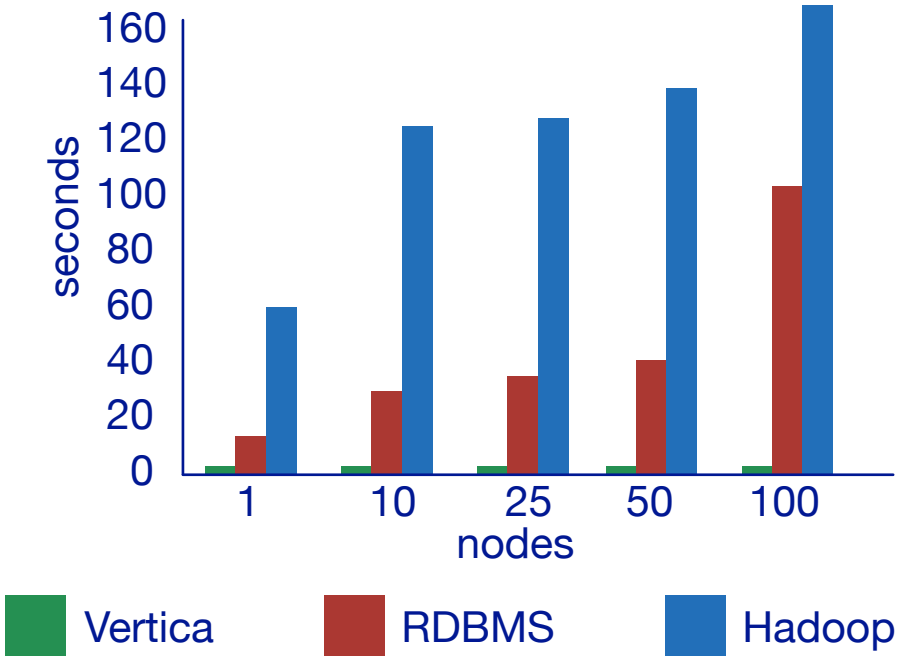
MapReduce vs. RDBMS: grep




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


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

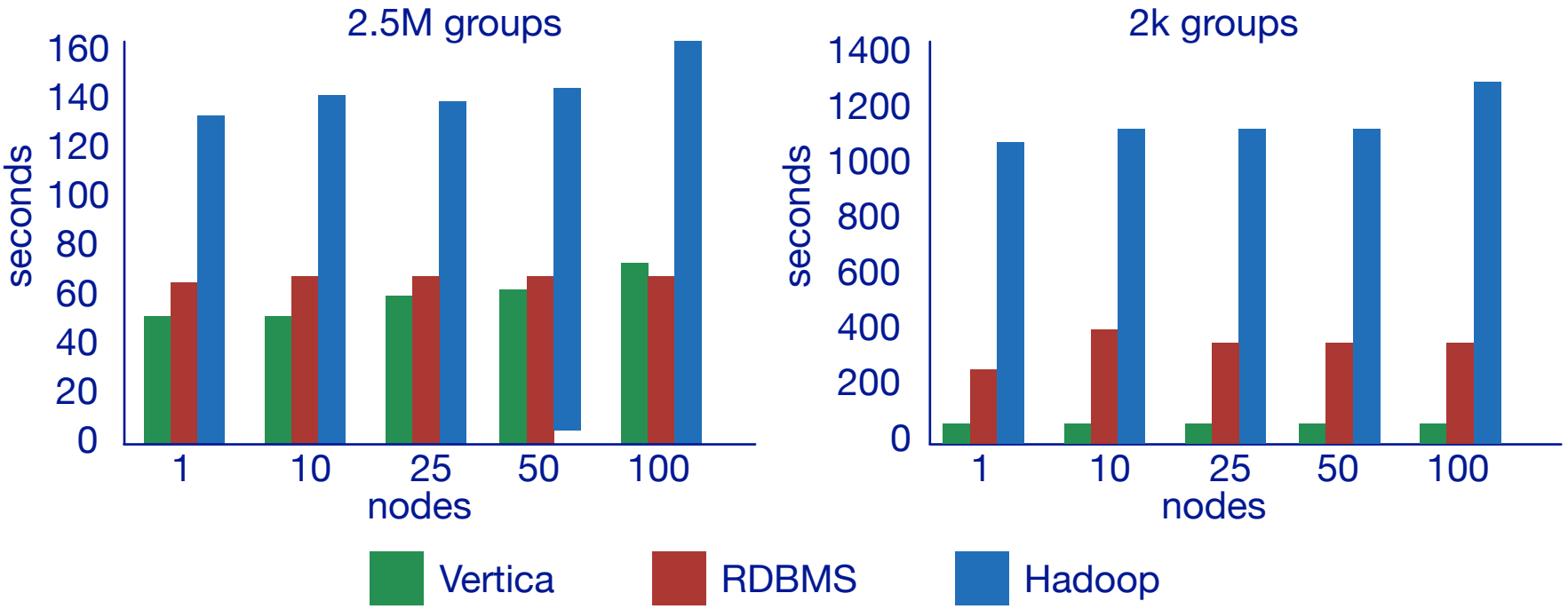
MapReduce vs. RDBMS: select




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


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

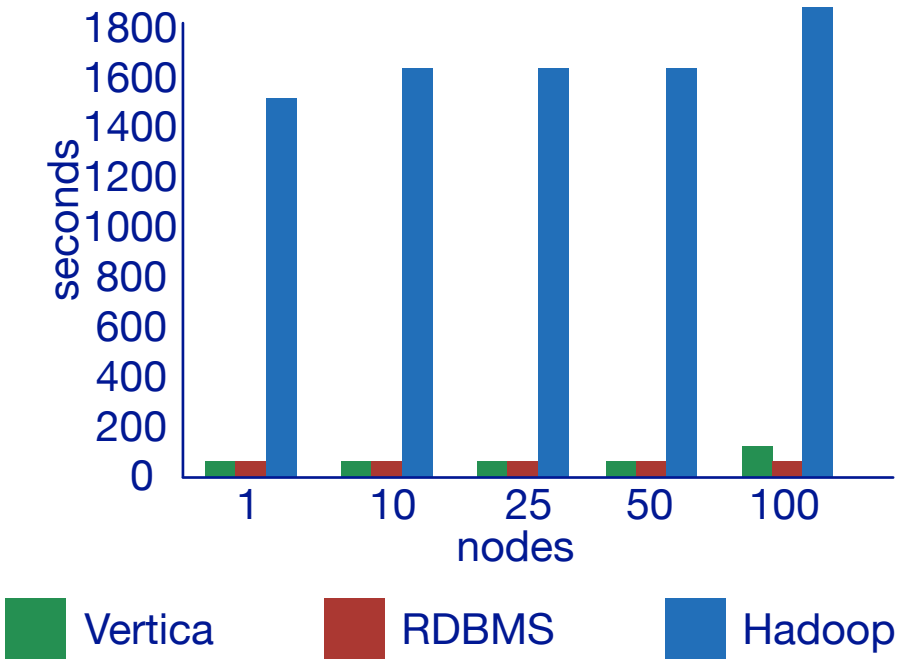
MapReduce vs. RDBMS: aggregation




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


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

MapReduce vs. RDBMS: join



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

Why?

- Schemas are a good idea
  - Parsing fields out of flat text files is slow
  - Schemas define a contract, decoupling logical from physical
- Schemas allow for building efficient auxiliary structures
  - Value indexes, join indexes, etc.
- Relational algorithms have been optimised for the underlying system
  - The system itself has complete control of performance-critical decisions
  - Storage layout, choice of algorithm, order of execution, etc.

Why?

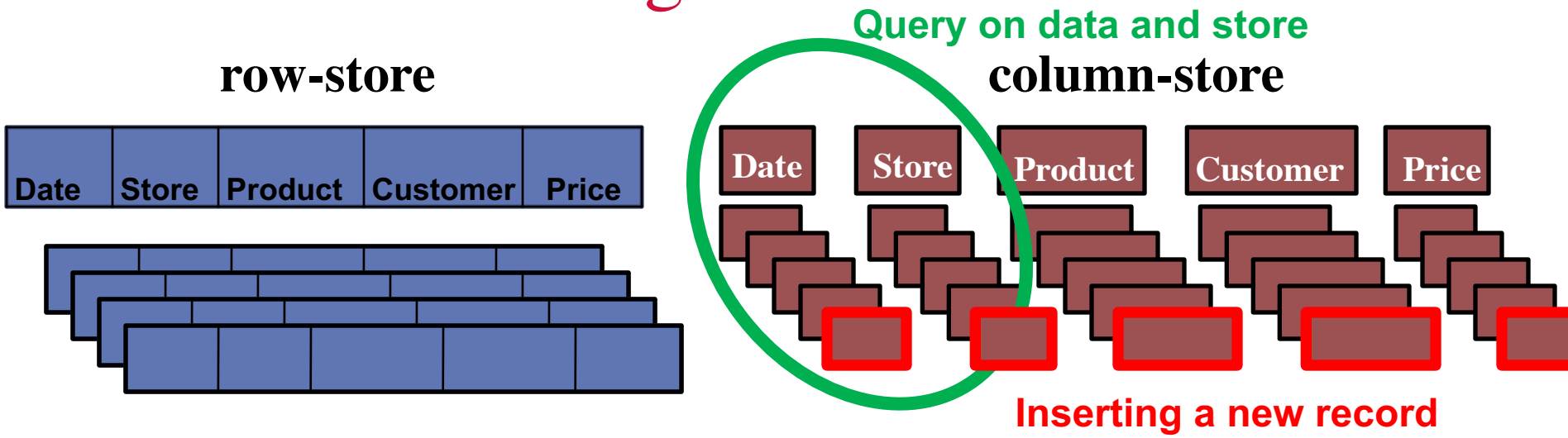
- Schemas are a good idea
  - Parsing fields out of flat text files is slow
  - Schemas define a contract, decoupling logical from physical
- Schemas allow for building efficient auxiliary structures
  - Value indexes, join indexes, etc.
- Relational algorithms have been optimised for the underlying system
  - The system itself has complete control of performance-critical decisions
  - Storage layout, choice of algorithm, order of execution, etc.

Analytical SQL data systems

Important architectural dimensions

- Storage
  - **columnar storage**
  - data compression
  - data pruning
  - table partitioning & distribution
- Query execution
  - vectorized execution
  - and/or.. JIT code generation
  - query optimization
  - update infrastructure

Columnar Storage



+ easy to add/modify a record

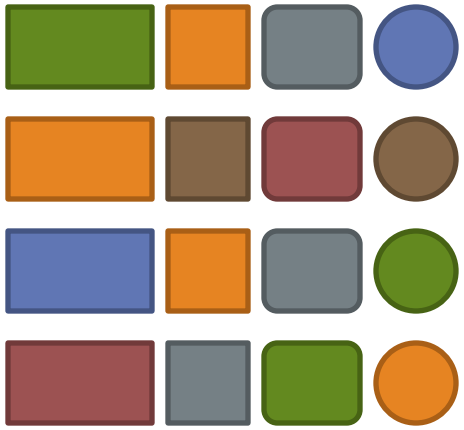
- might read in unnecessary data

+ only need to read in relevant data

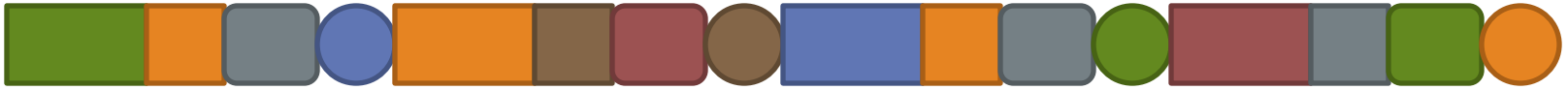
- tuple writes require multiple accesses

=> *suitable for read-mostly, read-intensive, large data repositories*

Storage layout: row vs. column stores



Row store



Column store

Storage layout: row vs. column stores

- Row stores
  - Easy to modify a record
  - Might read unnecessary data when processing
- Column stores
  - Only read necessary data when processing
  - Tuple writes require multiple accesses

Analytical SQL data systems

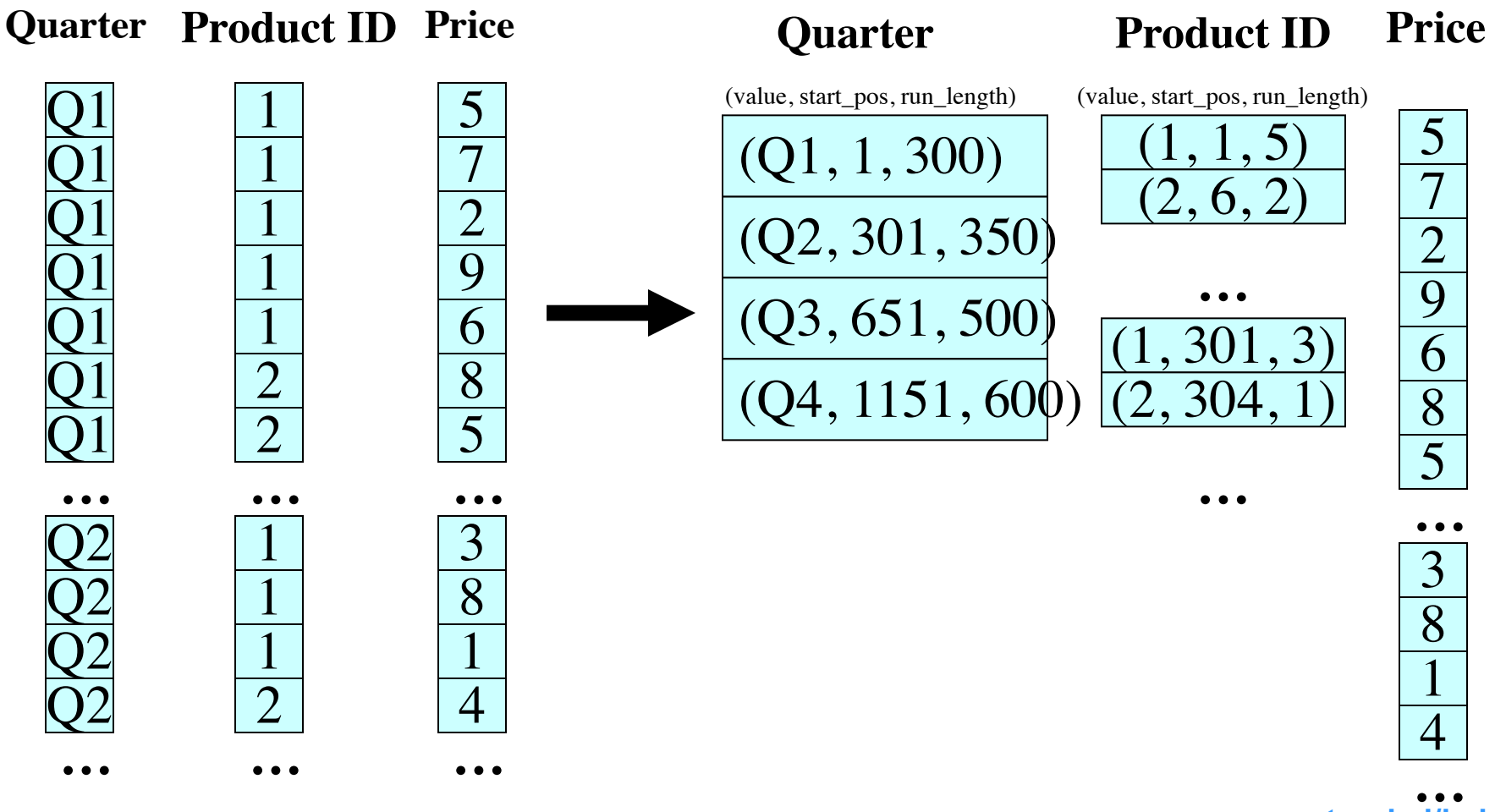
Important architectural dimensions

- Storage
  - columnar storage
  - **data compression**
  - data pruning
  - table partitioning & distribution
- Query execution
  - vectorized execution
  - and/or.. JIT code generation
  - update infrastructure

Columnar Compression

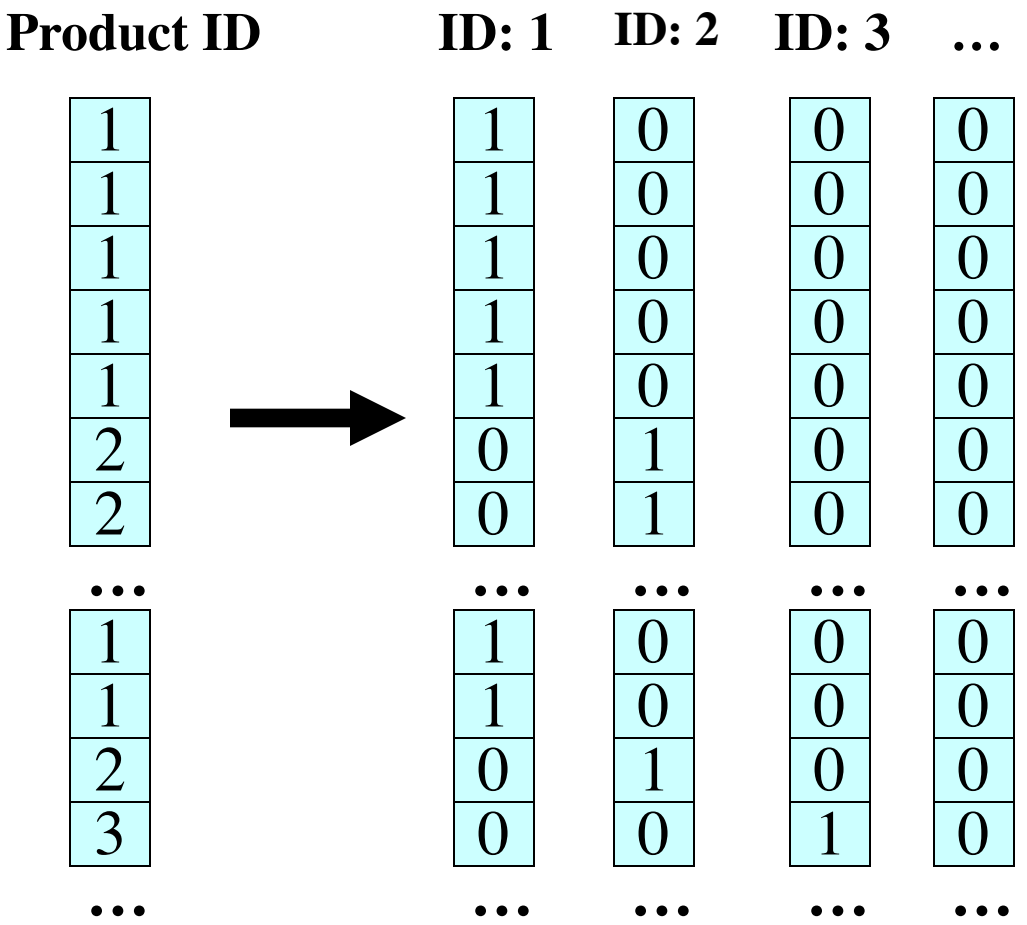
- Trades I/O for CPU
  - A winning proposition currently
  - Even trading RAM bandwidth for CPU wins
    - 64 core machines starved for RAM bandwidth
- Additional column-store synergy:
  - Column store: data of the same distribution close together
    - Better compression rates
    - Generic compression (gzip) vs Domain-aware compression
  - Synergy with vectorized processing (see later) compress/decompress/execution, SIMD
  - Can use extra space to store multiple copies of data in different sort orders (see later)

Run-length Encoding



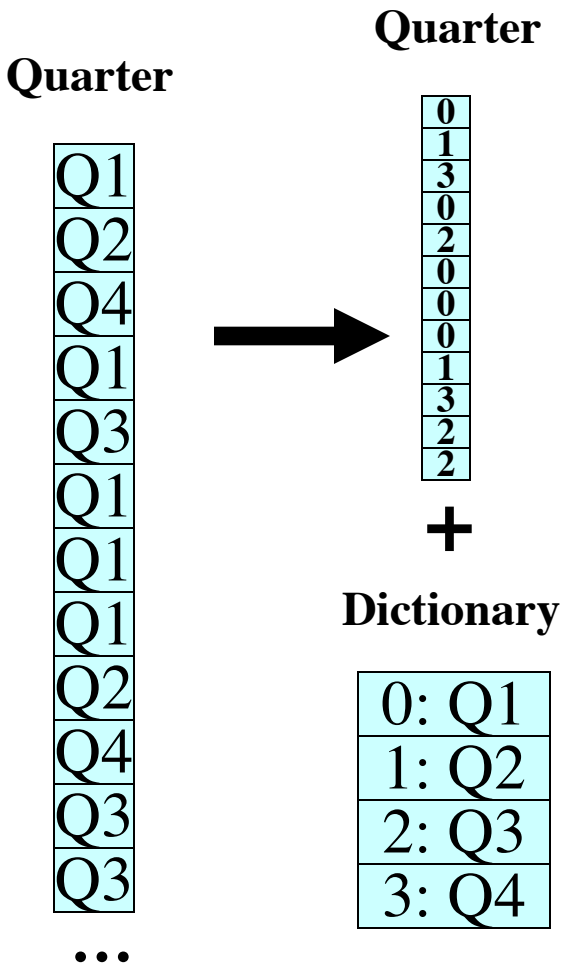
Bitmap Encoding

- For each unique value, v, in column c, create bit-vector b
  - b[i] = 1 if c[i] = v
- Good for columns with few unique values
- Each bit-vector can be further compressed if sparse



Dictionary Encoding

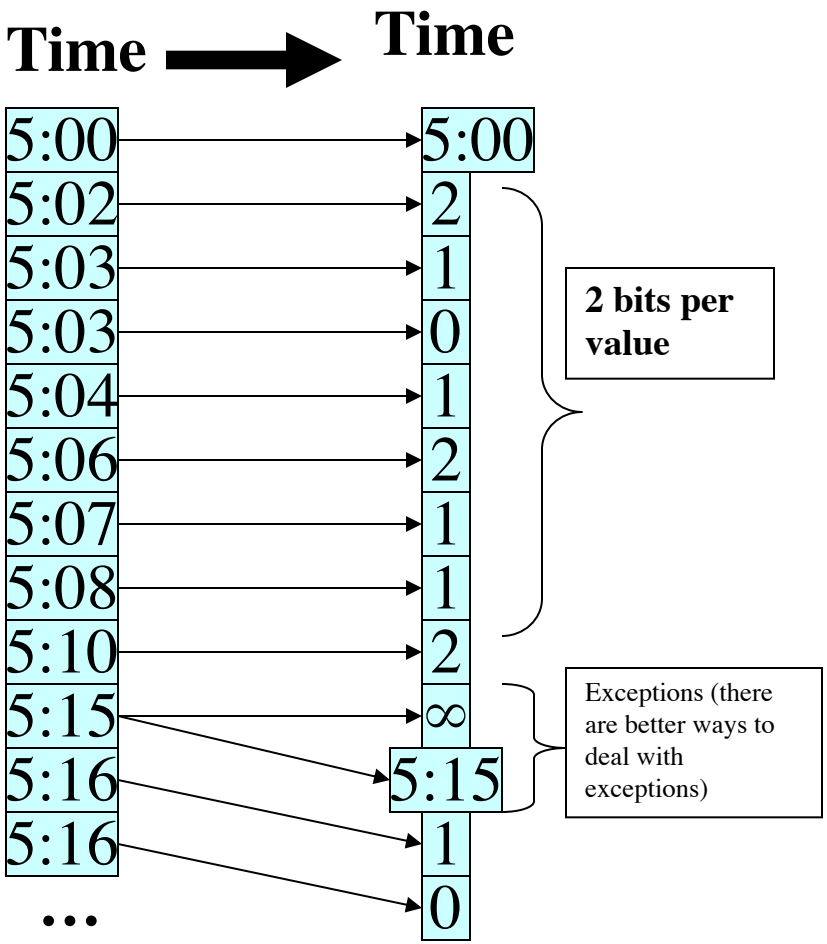
- For each unique value create dictionary entry
- Dictionary can be per-block or per-column
- Column-stores have the advantage that dictionary entries may encode multiple values at once



Differential Encoding

- Encodes values as b bit offset from previous value
- Special escape code (just like frame of reference encoding) indicates a difference larger than can be stored in b bits
  - After escape code, original (uncompressed) value is written
- Performs well on columns containing increasing/decreasing sequences
  - inverted lists
  - timestamps
  - object IDs
  - sorted / clustered columns

"Improved Word-Aligned Binary Compression for Text Indexing" Ahn, Moffat, TKDE'06



Operating Directly on Compressed Data

Examples

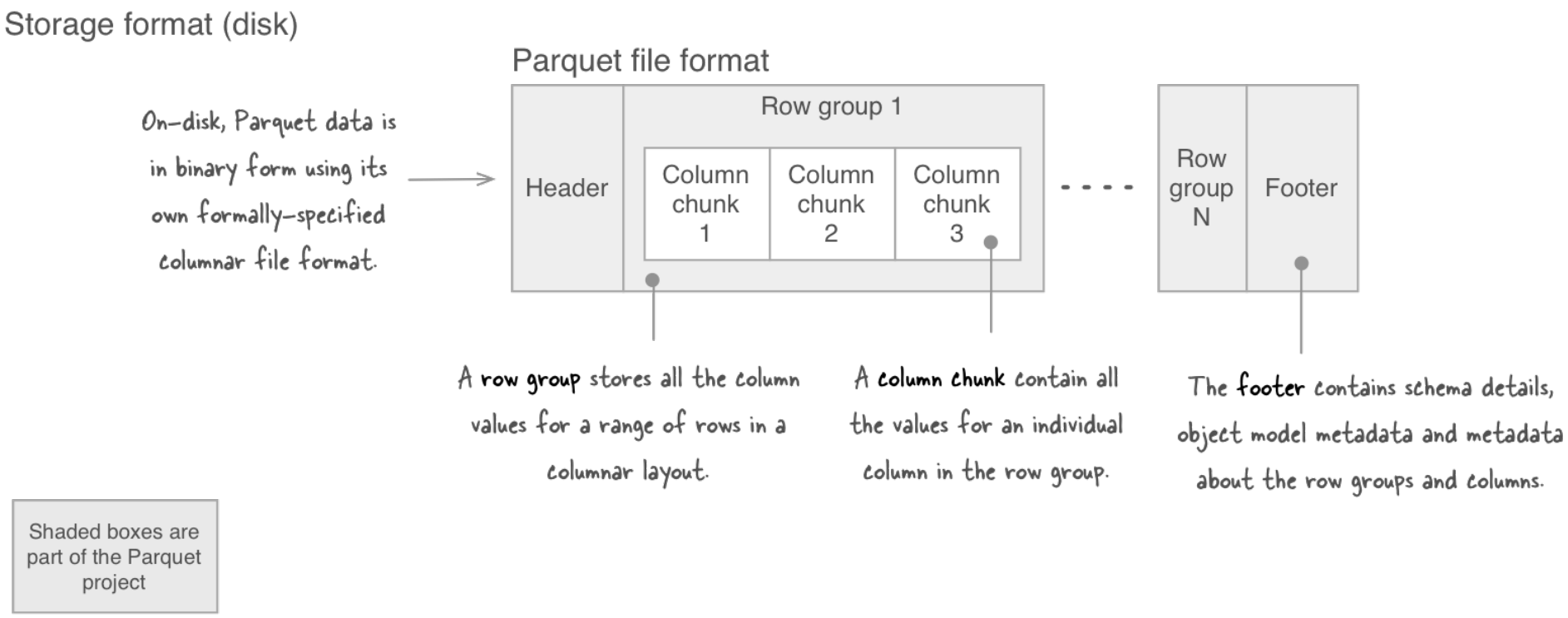
- **SUM_i(rle-compressed column[i]) ➔ SUM_g(count[g] * value[g])**
- **(country == "France") ➔ countryCode == 6**

strcmp **SIMD**

Benefits:

- **I/O - CPU tradeoff is no longer a tradeoff (CPU also gets improved)**
- **Reduces memory–CPU bandwidth requirements**
- **Opens up possibility of operating on multiple records at once**

Example: Parquet Format



Parquet Format

Most popular compressed columnar format nowadays

- In the cloud, but also on Hadoop
- Organizes data in row groups (say: 100MB table chunks)
- Each row group consist of data chunks
  - One chunk per column
  - Allows to skip vertically (unused columns)
- Data is compressed
  - Combination of dictionary, bit-packing and RLE encoding
  - + general-purpose compression on top (gzip, snappy or LZ4)
- MinMax Indexes for better pruning
  - pruning = skipping horizontally

Pruning with MinMax

- Data is often naturally ordered
  - very often, on date
- Data is often correlated
  - orderdate/paydate/shipdate
  - marketing campaigns/date
  - ..correlation is everywhere

    ..hard to predict

Zone Maps

- Very sparse index
- Keeps MinMax for every column
- Cheap to maintain
  - Just widen bounds on each modification

Q: acctno BETWEEN 150 AND 200?

Accounts

KEY	acctno	name	balance	
00	019	Isabella	269.38	zone 0
01	038	Jackson	914.11	
02	072	Lucas	346.61	
03	156	Sophia	266.55	
04	153	Mason	850.90	zone 1
05	282	Ethan	521.60	
06	389	Emily	647.38	
07	314	Lily	119.40	
08	332	Chloe	526.08	zone 2
09	302	Emma	497.19	
10	533	Aiden	22.03	
11	592	Ava	140.67	
12	808	Mia	383.69	zone 3
13	896	Jacob	899.41	

Accounts.MinMax

zone	KEY		acctno		name		balance	
	min	max	min	max	min	max	min	max
0	00	03	019	156	Isabella	Sophia	266.55	914.11
1	04	07	153	389	Emily	Mason	119.40	850.90
2	08	11	332	592	Aiden	Emma	22.03	526.08
3	12	13	808	896	Mia	Jacob	383.69	899.41

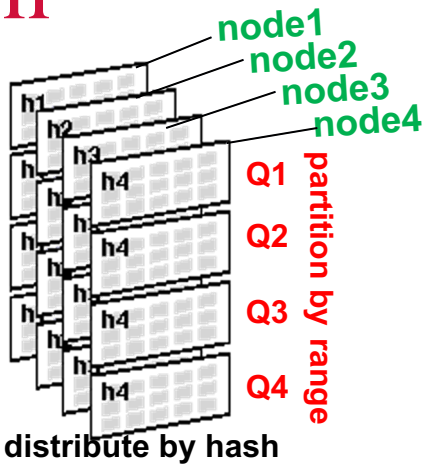
Q: key BETWEEN 13 AND 15?

Analytical SQL data systems

Important architectural dimensions

- Storage
  - columnar storage
  - data compression
  - **data pruning**
  - **table partitioning & distribution**
- Query execution
  - vectorized execution
  - and/or.. JIT code generation
  - update infrastructure

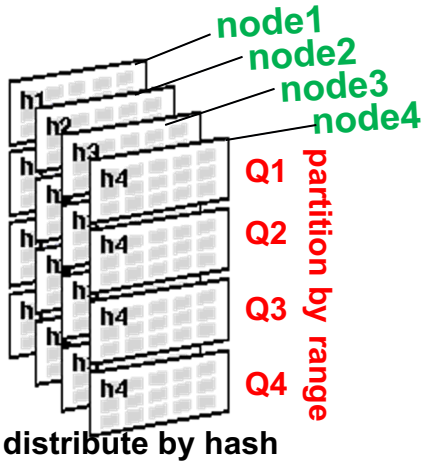
Table Partitioning and Distribution



- data is spread based on a Key
  - Functions: Hash, Range, List
- "distribution"
  - Goal: parallelism
    - give each compute node a piece of the data
    - each query has work on every piece (keep everyone busy)
- "partitioning"
  - Goal: data lifecycle management
    - Data warehouse e.g. keeps last six months
    - Every night: load one new day, drop the oldest partition
  - Goal: improve access pattern
    - when querying for May, drop Q1,Q3,Q4 ("partition pruning")

Which kind of function would you use for which method?

Data Placement in Hadoop



- Each node writes the partitions it owns
  - Where does the data end up, really?
- HDFS default block placement strategy:
  - Node that initiates writes gets first copy
  - 2nd copy on the same rack
  - 3rd copy on a different rack
- Rows from the same record should on the same node
  - Not entirely trivial in column stores
    - Column partitions should be co-located
  - Simple solution:
    - Put all columns together in one file (e.g. Parquet, or ORC)
  - Complex solution:
    - Replace the default HDFS block placement strategy by a custom one

Data Placement in the Cloud

- There is no data locality in the cloud!
  - To create elasticity, compute is de-coupled from storage
  - i.e. S3 files are always coming from remote
- distribution & partitioning is *very* common
  - A table is buckets full of thousands or millions S3 files
  - Goals: work distribution & lifecycle management (again)
- some locality can be created by caching
  - Caching in memory (Spark)
  - Caching on local ephemeral disk (e.g. DBIO cache in Databricks)

Analytical SQL data systems

Important architectural dimensions

- Storage
  - columnar storage
  - data compression
  - data pruning
  - table partitioning & distribution
- **Query execution**
  - vectorized execution
  - and/or.. JIT code generation
  - update infrastructure

Analytical SQL data systems

Important architectural dimensions

- Storage
  - columnar storage
  - data compression
  - data pruning
  - table partitioning & distribution
- **Query execution**
  - vectorized execution
  - and/or.. JIT code generation
  - update infrastructure

DBMS Computational Efficiency?

TPC-H 1GB, query 1

- selects 98% of fact table, computes net prices and aggregates all
- Results:
  - C program: ?
  - MySQL: 26.2s
  - DBMS “X”: 28.1s

“MonetDB/X100: Hyper-Pipelining Query Execution ” Boncz, Zukowski, Nes, CIDR’05

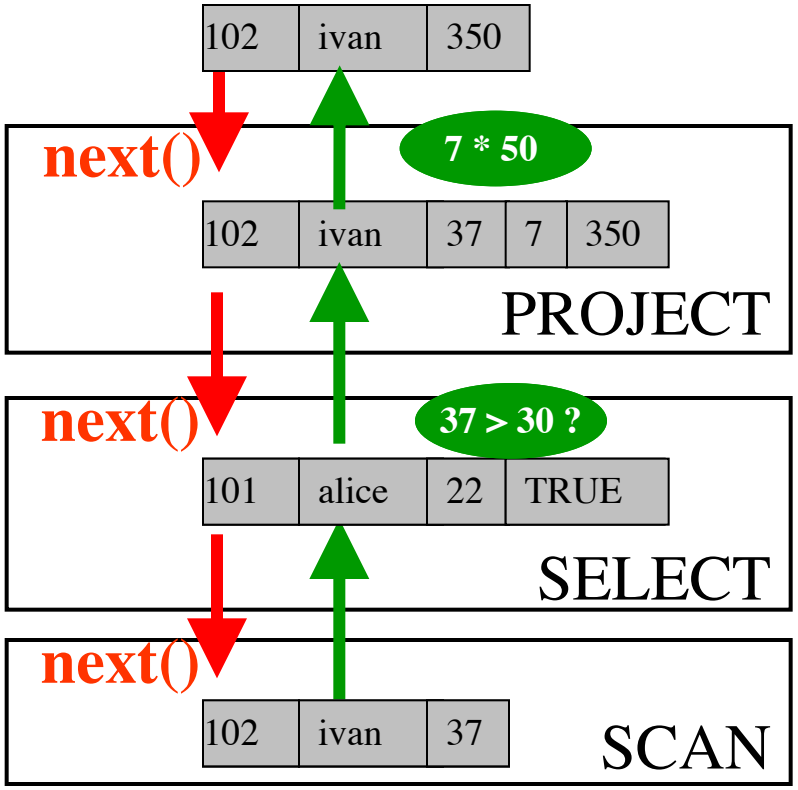
DBMS Computational Efficiency?

TPC-H 1GB, query 1

- selects 98% of fact table, computes net prices and aggregates all
- Results:
  - C program: **0.2s**
  - MySQL: 26.2s
  - DBMS “X”: 28.1s

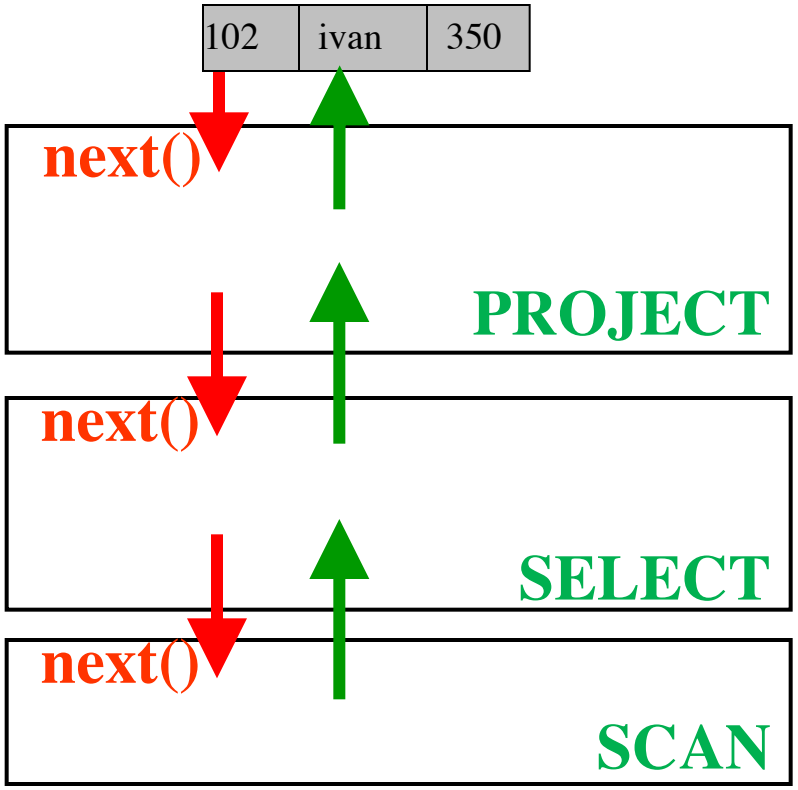
“MonetDB/X100: Hyper-Pipelining Query Execution ” Boncz, Zukowski, Nes, CIDR’05

How Do Query Engines Work?




```
SELECT   id, name
         (age-30)*50 AS bonus
FROM     employee
WHERE    age > 30
```

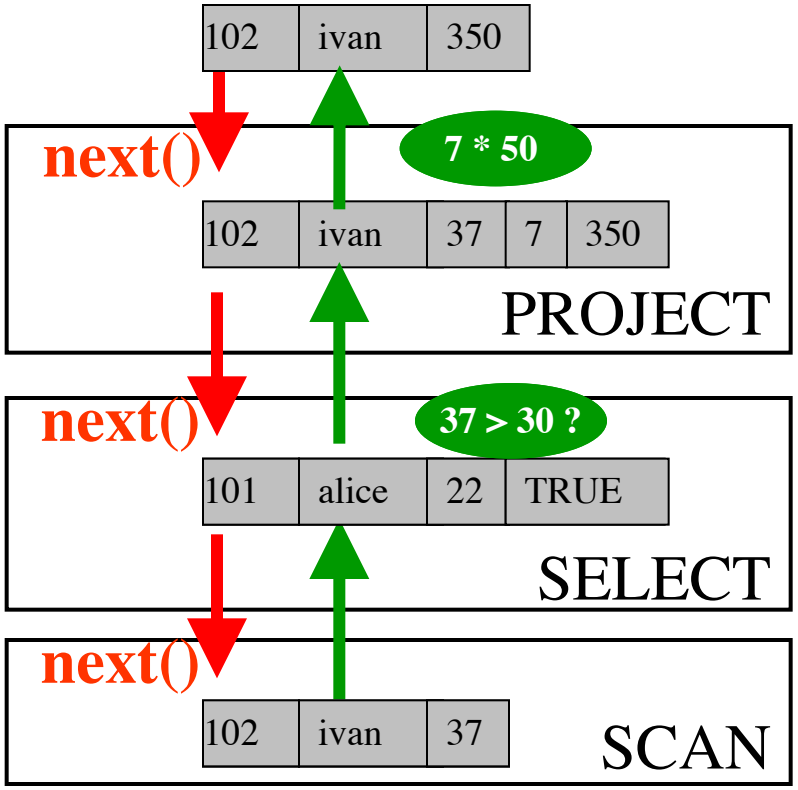
How Do Query Engines Work?



Operators

Iterator interface
-open()
-next(): tuple
-close()

How Do Query Engines Work?



Primitives

Provide computational functionality

All arithmetic allowed in expressions,
e.g. Multiplication

7 * 50

`mult(int,int)` ➔ `int`

Analytical SQL data systems

Important architectural dimensions

- Storage
  - columnar storage
  - data compression
  - data pruning
  - table partitioning & distribution
- **Query execution**
  - **vectorized execution**
  - and/or.. JIT code generation
  - update infrastructure

"MonetDB/X100: Hyper-Pipelining Query Execution" Boncz, Zukowski, Nes, CIDR'05

vectorwise

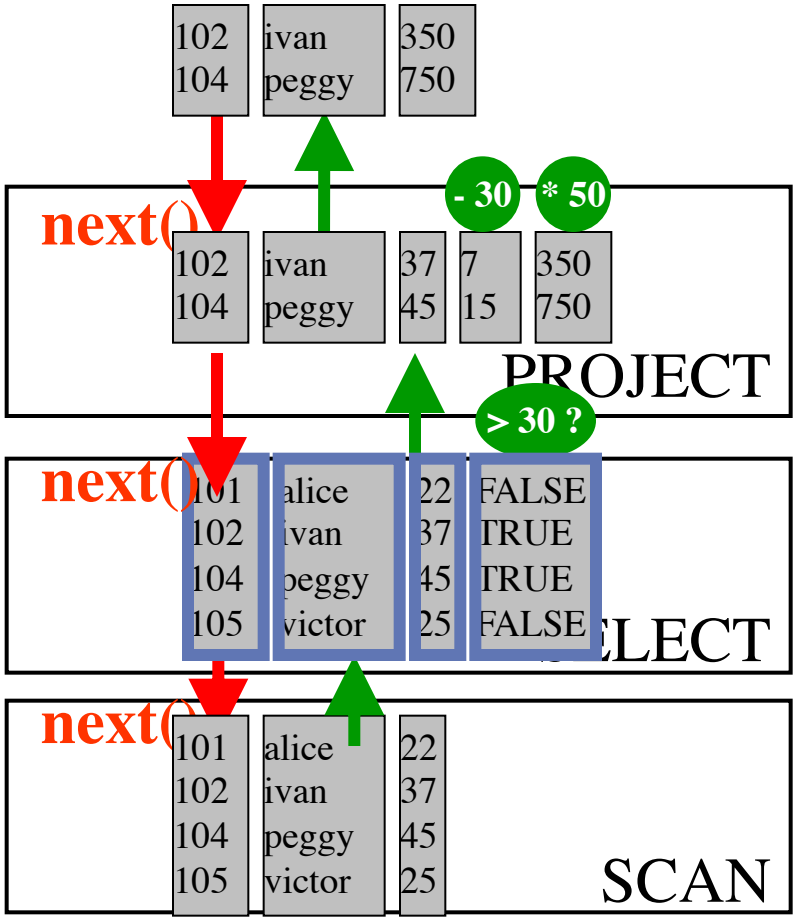
Observations:

"Vectorized In Cache Processing"

vector = array of ~100

processed in a tight loop

CPU cache Resident



"MonetDB/X100: Hyper-Pipelining Query Execution" Boncz, Zukowski, Nes, CIDR'05

vectorwise

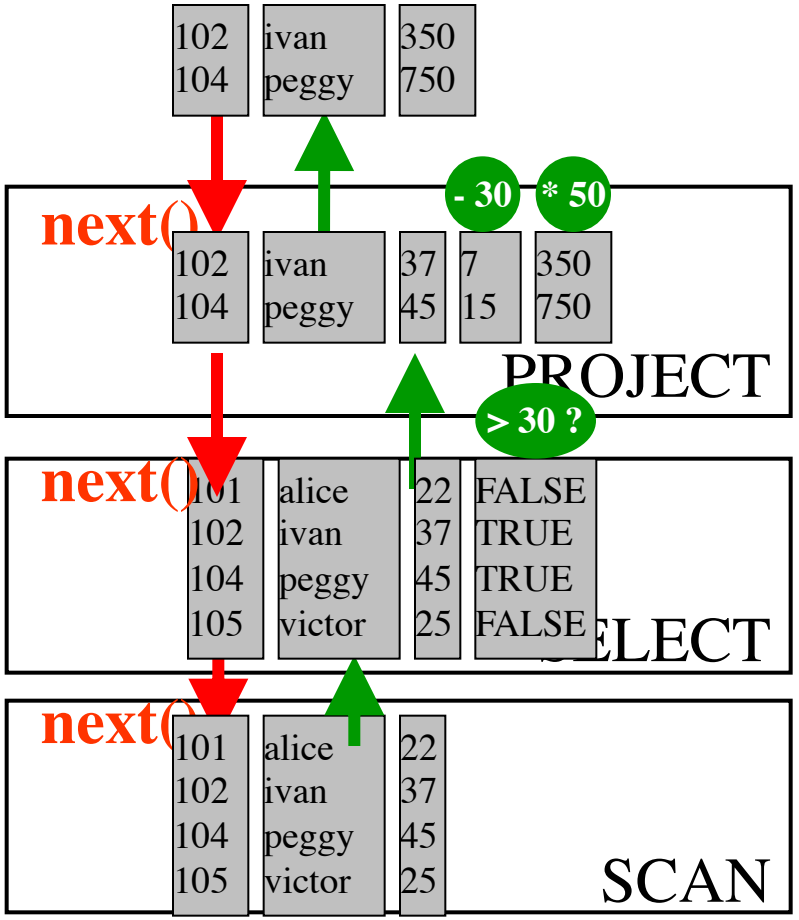
Observations:

next() called much less often ➔ more time spent in primitives less in overhead

primitive calls process an

CPU Efficiency depends on "nice" code
- out-of-order execution
- few dependencies (control,data)
- compiler support

Compilers like simple loops over arrays
- loop-pipelining
- automatic SIMD



vectorwise

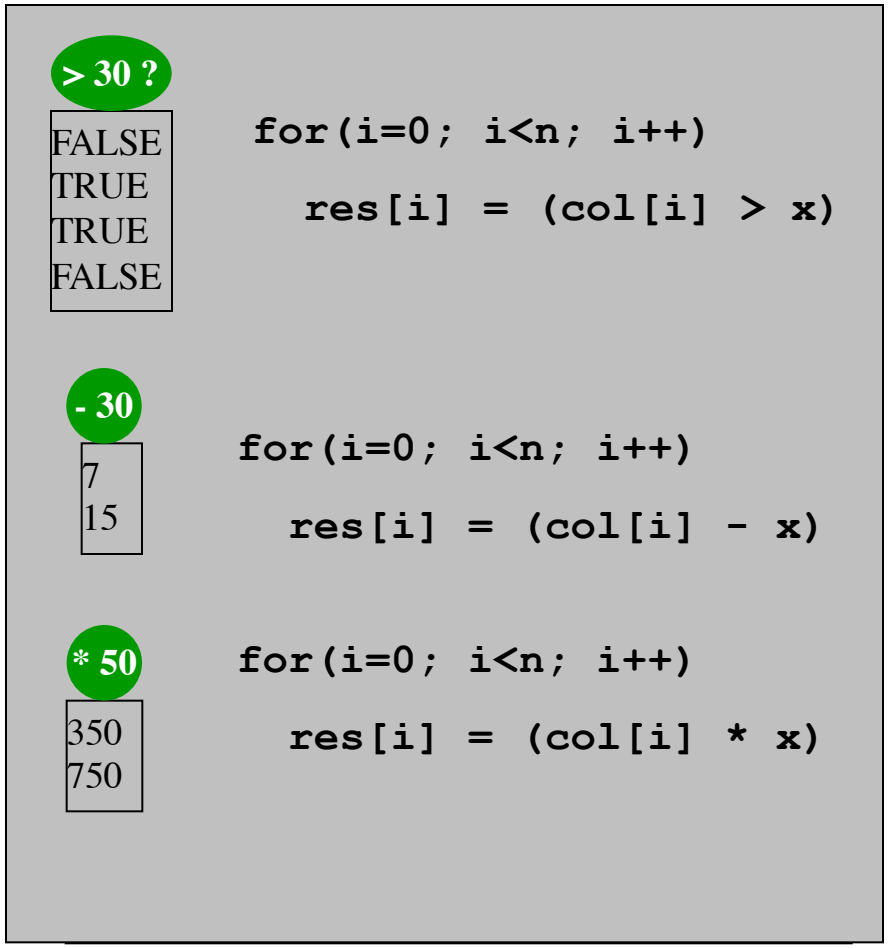
Observations:

next() called much less often ➔ more time spent in primitives less in overhead

primitive calls process an

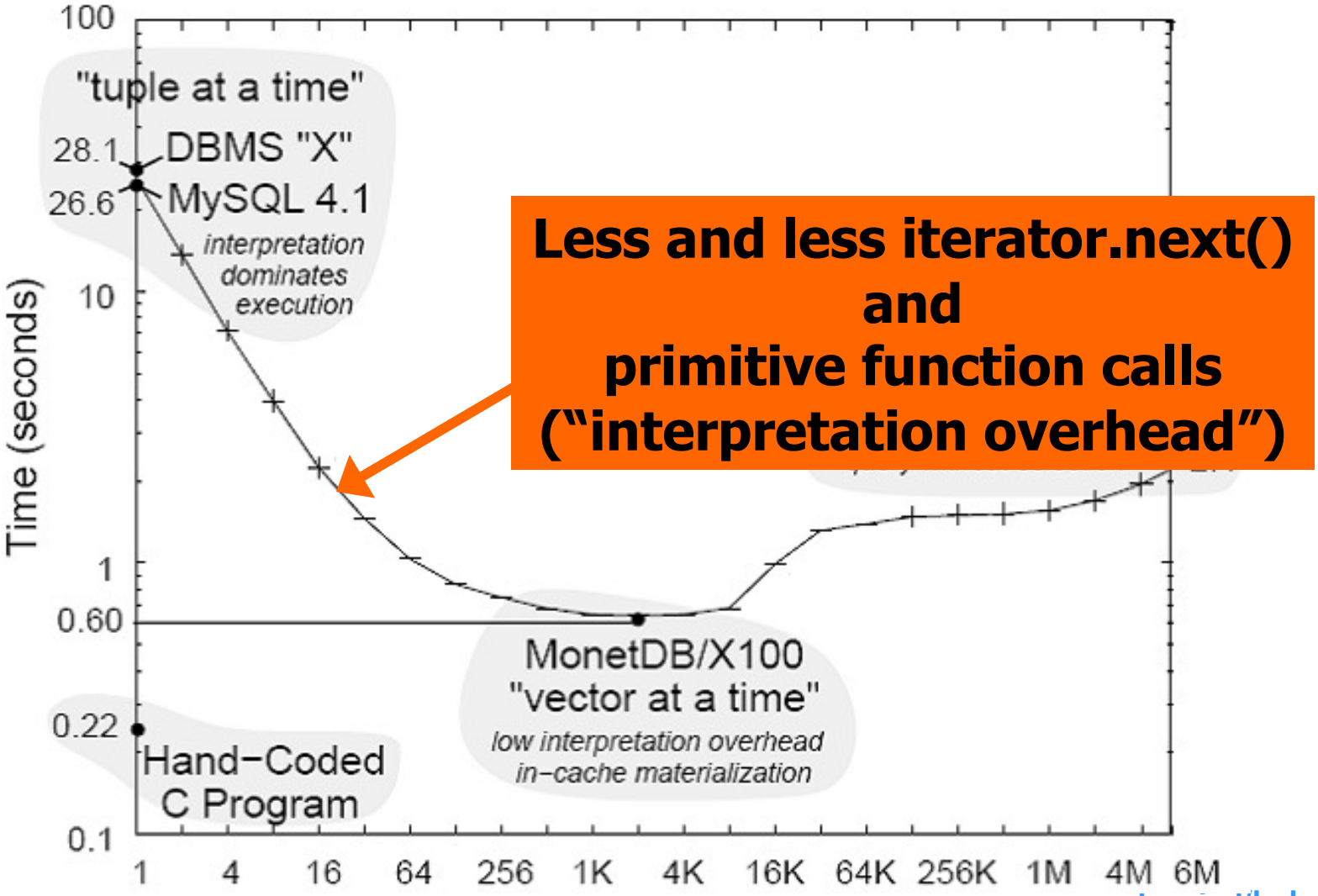
CPU Efficiency depends on "nice" code
- out-of-order execution
- few dependencies (control,data)
- compiler support

Compilers like simple loops over arrays
- loop-pipelining
- automatic SIMD



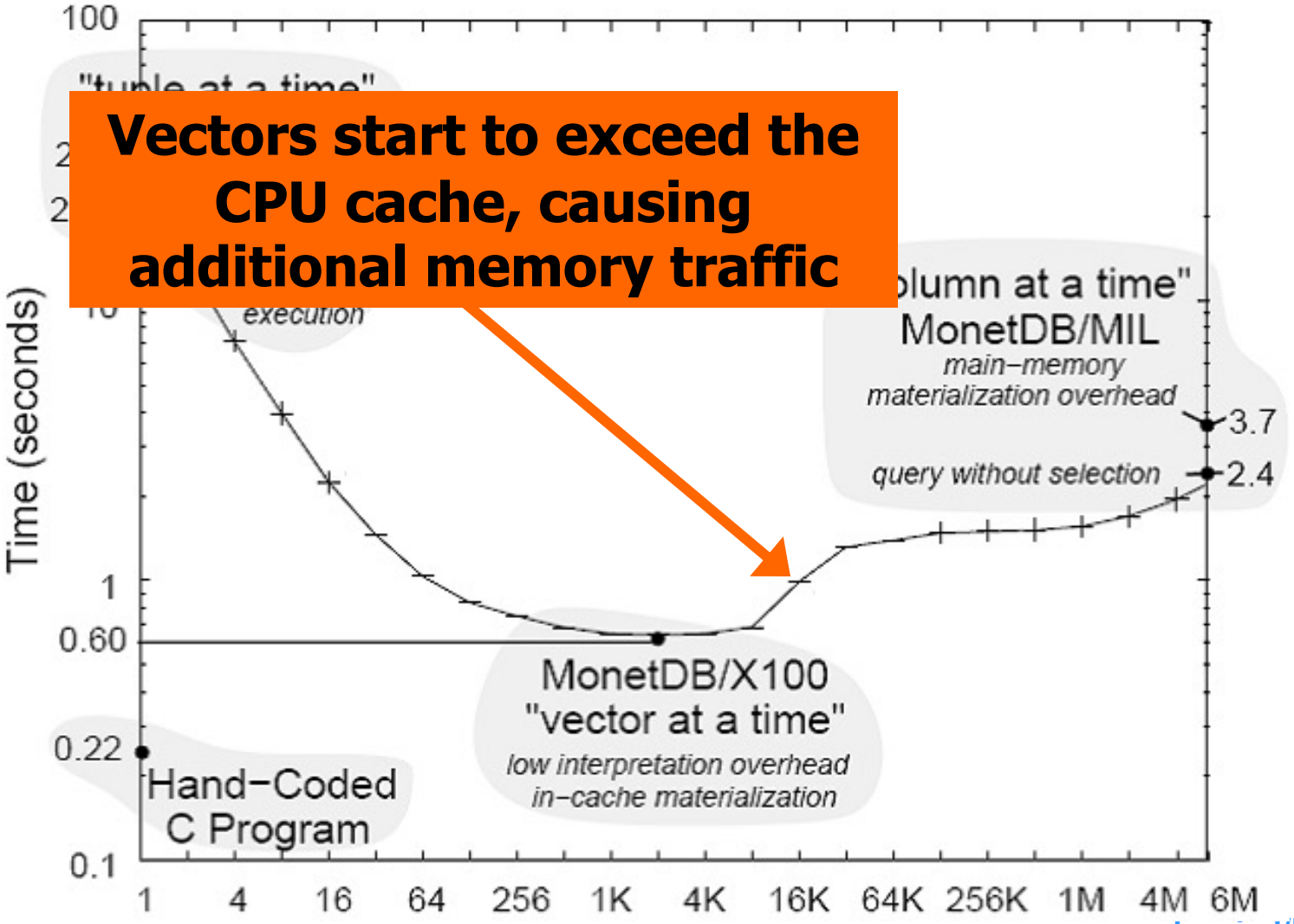
"MonetDB/X100: Hyper-Pipelining Query Execution " Boncz, Zukowski, Nes, CIDR'05

Varying the Vector size



"MonetDB/X100: Hyper-Pipelining Query Execution " Boncz, Zukowski, Nes, CIDR'05

Varying the Vector size



Analytical SQL data systems

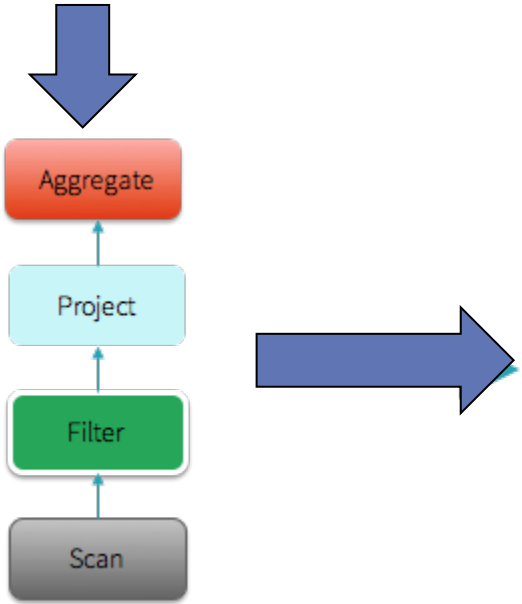
Important architectural dimensions

- Storage
  - columnar storage
  - data compression
  - data pruning
  - table partitioning & distribution
- **Query execution**
  - vectorized execution
  - and/or.. **JIT code generation**
  - update infrastructure

Just In Time (JIT) query compilation

- Automatically generate a **program** that executes the query
  - like a hand-written program that solves the task
  - No-frills, no overheads, tight for-loops
- Used in:
  - HyPer (Tableau), Impala (SQL ➔ LLVM IR)
  - Spark (SQL ➔ java) , Redshift (SQL ➔ C++)

```
SELECT count(*) FROM sales WHERE ss_item_sk = 1000
```




```
long count = 0;
for (ss_item_sk in store_sales) {
  if (ss_item_sk == 1000) {
    count += 1;
  }
}
```

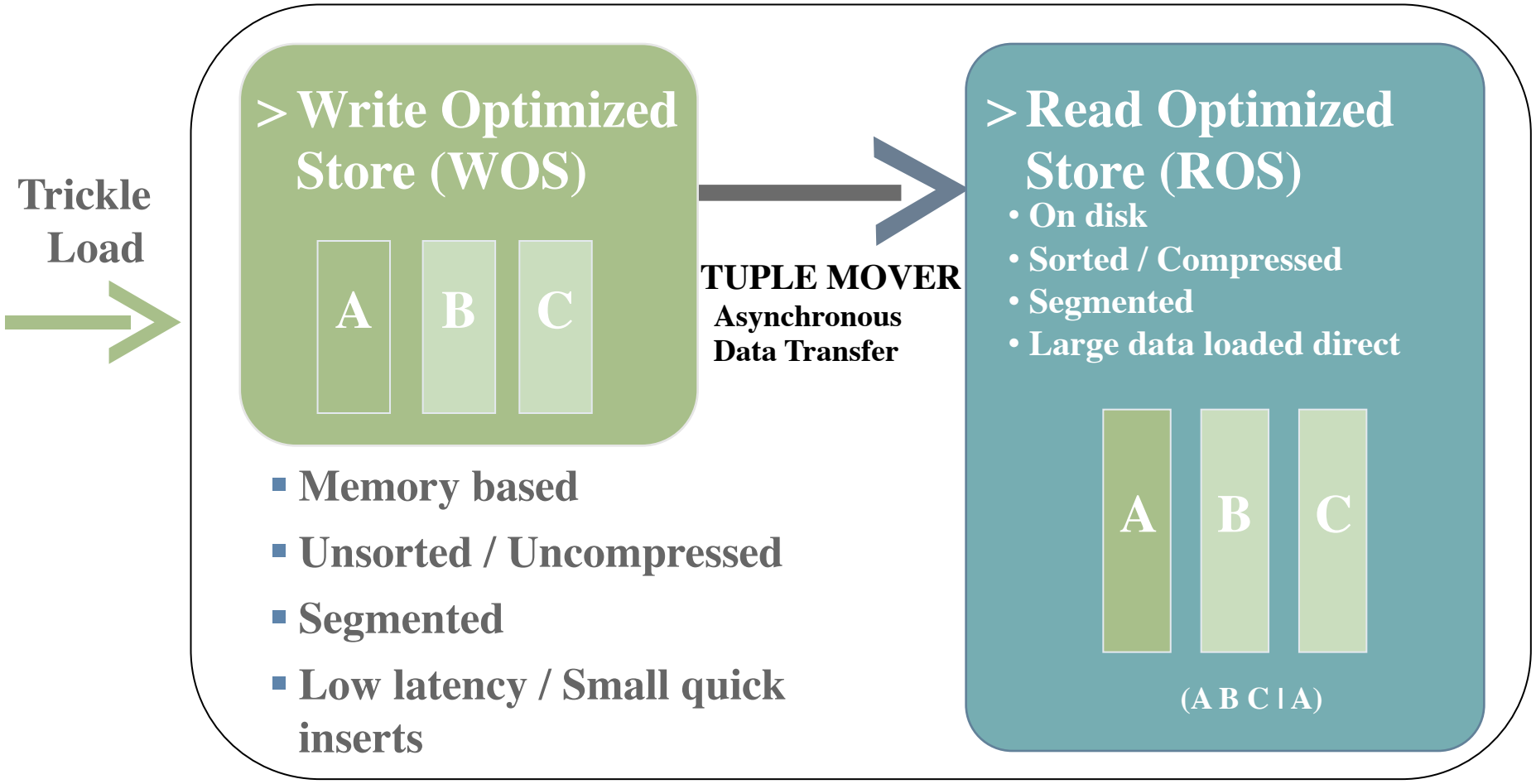
Analytical SQL data systems

Important architectural dimensions

- Storage
  - columnar storage
  - data compression
  - data pruning
  - table partitioning & distribution
- **Query execution**
  - vectorized execution
  - and/or.. JIT code generation
  - **update infrastructure**

Batch Update Infrastructure (Vertica)

Challenge: hard to update columnar compressed data



Batch Update Infrastructure (Hive)

Challenge: HDFS read-only + large block size

Base File

Name	Purchase
Anne	Red Fish
Bill	Blue Fish
Christine	Blue Fish
David	Black Fish
Eric	Young Fish

Update 1

Op	Txn Id	RowId	Name	Purchase
I	1	0	Joe	Red Fish
U	0	0	Anne	Star
D	0	4		

Update 2

Op	Txn Id	RowId	Name	Purchase
U	1	0	Joe	Old Fish
U	0	0	Ann	Star
D	0	2		

Merge During Query Processing

Logical File

Name	Purchase
Joe	Old Fish
Ann	Star
Bill	Blue Fish
David	Black Fish

Hortonworks

Batch Update Infrastructure (Hive)

Challenge: HDFS read-only + large block size

Base File

Name	Purchase
Anne	Red Fish
Bill	Blue Fish
Christine	Blue Fish
David	Black Fish
Eric	Young Fish

Update 1

Op	Txn Id	RowId	Name	Purchase
I	1	0	Joe	Red Fish
U	0	0	Anne	Star
D	0	4		

Update 2

Op	Txn Id	RowId	Name	Purchase
U	1	0	Joe	Old Fish
U	0	0	Ann	Star
D	0	2		

Merge During Query Processing

Logical File

Name	Purchase
Joe	Old Fish
Ann	Star
Bill	Blue Fish
David	Black Fish

Hortonworks

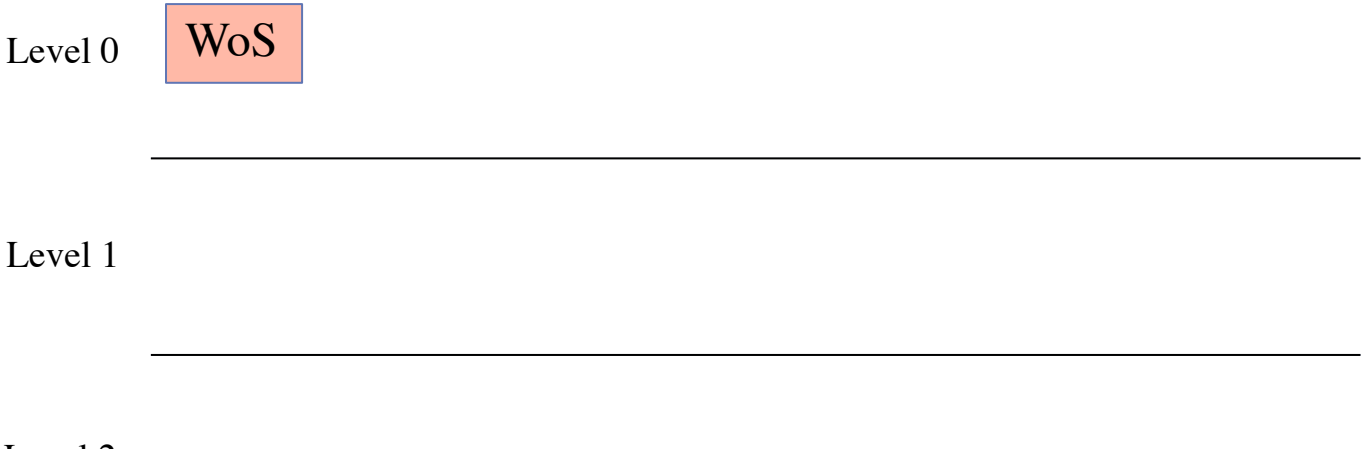
Batch Update Infrastructures

Conflicting concerns:

- Read-optimized data (compressed, columnar, large data units)
- …subjected to continuous update streams ➔ cannot update-in-place

Principle:

- Append updates to a write-optimized store (WoS) or logfile
- Regularly merge WoS with RoS (read-optimized store)



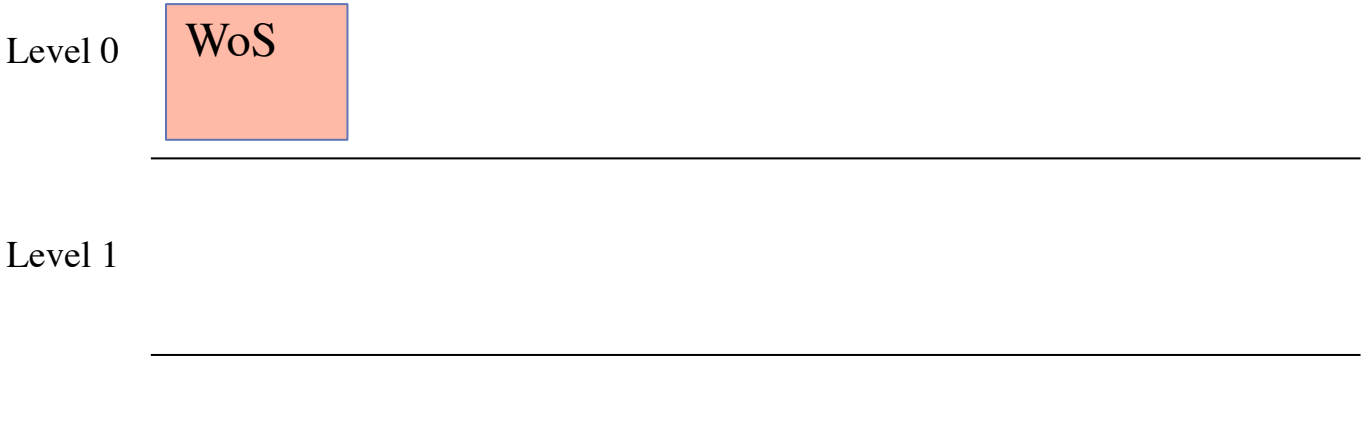
Batch Update Infrastructures

Conflicting concerns:

- Read-optimized data (compressed, columnar, large data units)
- …subjected to continuous update streams ➔ cannot update-in-place

Principle:

- Append updates to a write-optimized store (WoS) or logfile
- Regularly merge WoS with RoS (read-optimized store)



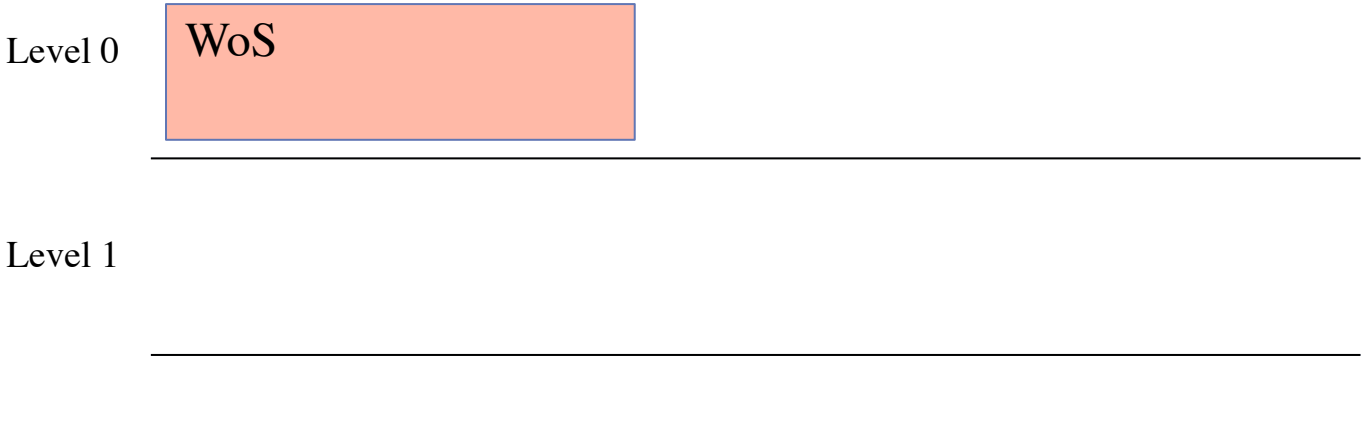
Batch Update Infrastructures

Conflicting concerns:

- Read-optimized data (compressed, columnar, large data units)
- …subjected to continuous update streams ➔ cannot update-in-place

Principle:

- Append updates to a write-optimized store (WoS) or logfile
- Regularly merge WoS with RoS (read-optimized store)



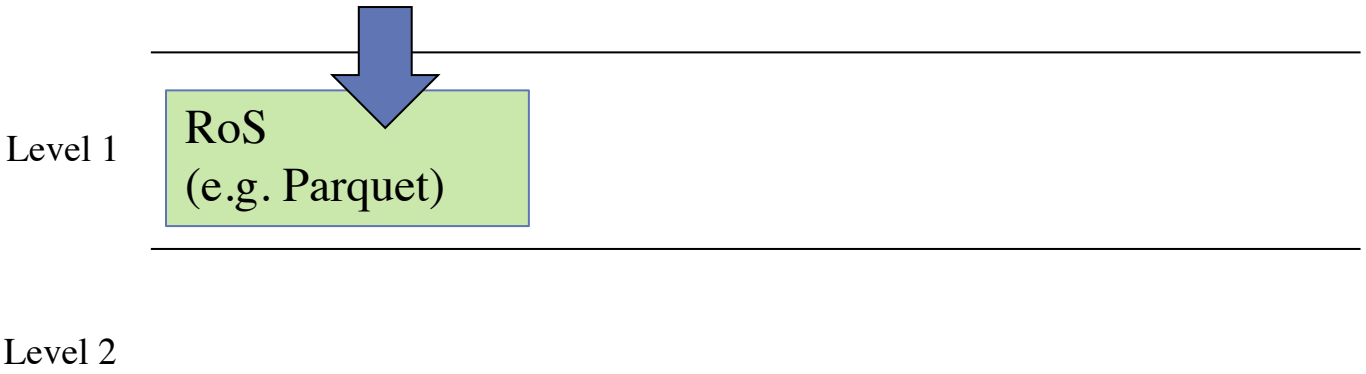
Batch Update Infrastructures

Conflicting concerns:

- Read-optimized data (compressed, columnar, large data units)
- …subjected to continuous update streams ➔ cannot update-in-place

Principle:

- Append updates to a write-optimized store (WoS) or logfile
- Regularly merge WoS with RoS (read-optimized store)



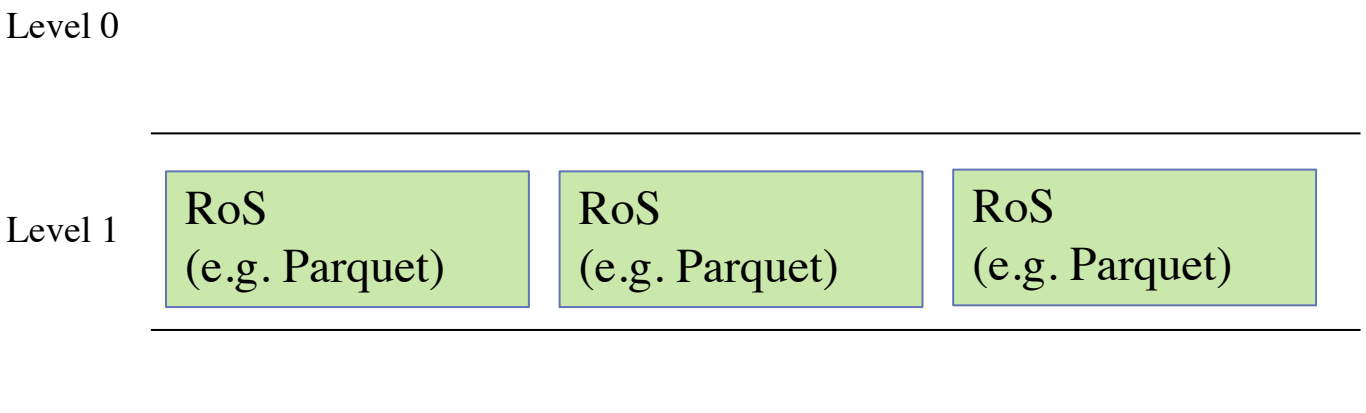
Batch Update Infrastructures

Conflicting concerns:

- Read-optimized data (compressed, columnar, large data units)
- …subjected to continuous update streams ➔ cannot update-in-place

Principle:

- Append updates to a write-optimized store (WoS) or logfile
- Regularly merge WoS with RoS (read-optimized store)



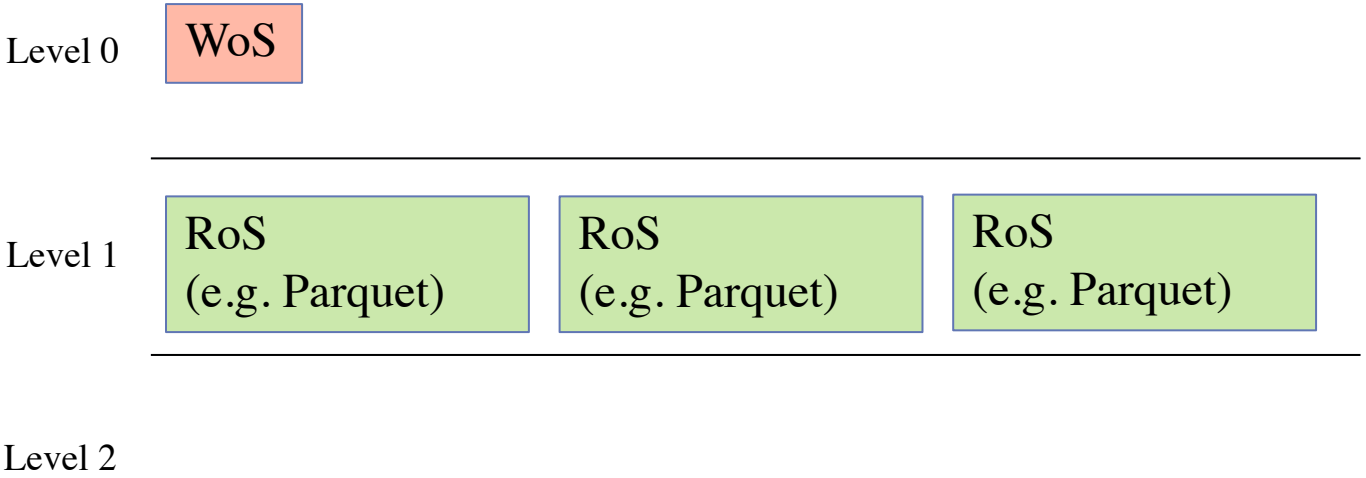
Batch Update Infrastructures

Conflicting concerns:

- Read-optimized data (compressed, columnar, large data units)
- …subjected to continuous update streams ➔ cannot update-in-place

Principle:

- Append updates to a write-optimized store (WoS) or logfile
- Regularly merge WoS with RoS (read-optimized store)



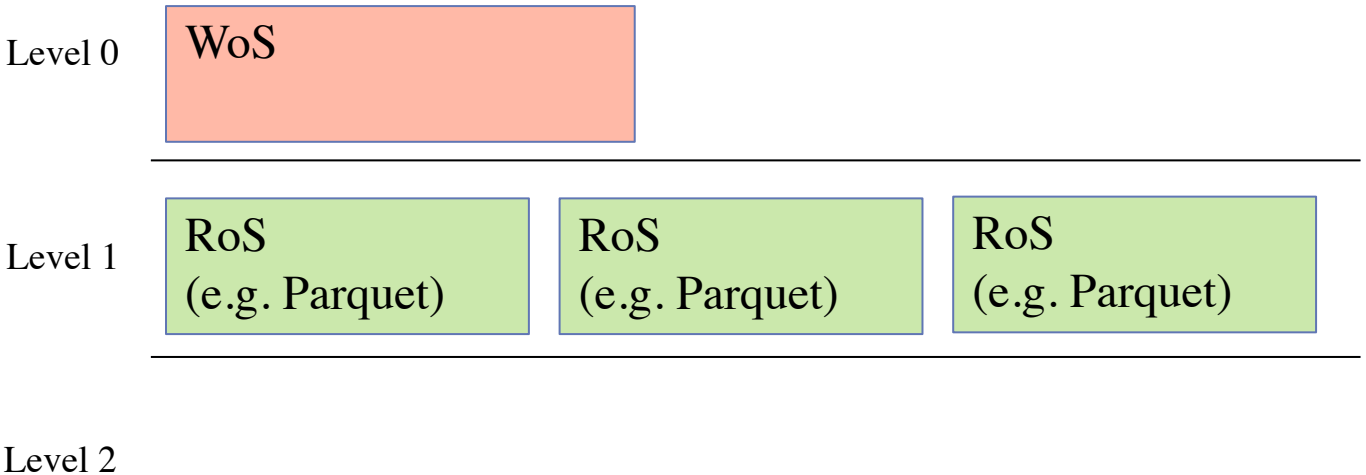
Batch Update Infrastructures

Conflicting concerns:

- Read-optimized data (compressed, columnar, large data units)
- …subjected to continuous update streams ➔ cannot update-in-place

Principle:

- Append updates to a write-optimized store (WoS) or logfile
- Regularly merge WoS with RoS (read-optimized store)



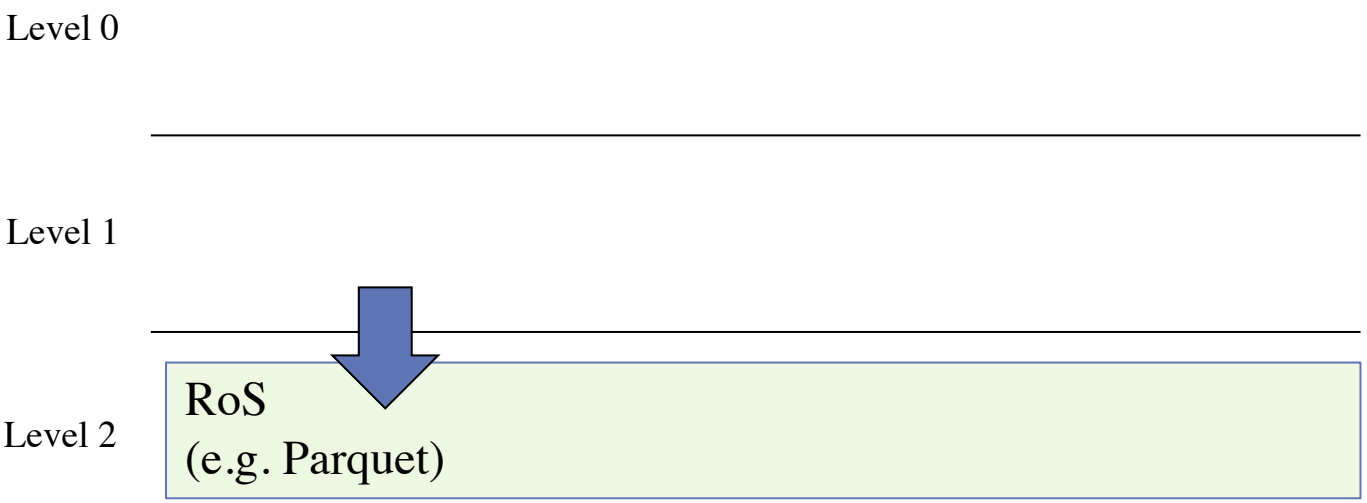
Batch Update Infrastructures

Conflicting concerns:

- Read-optimized data (compressed, columnar, large data units)
- …subjected to continuous update streams ➔ cannot update-in-place

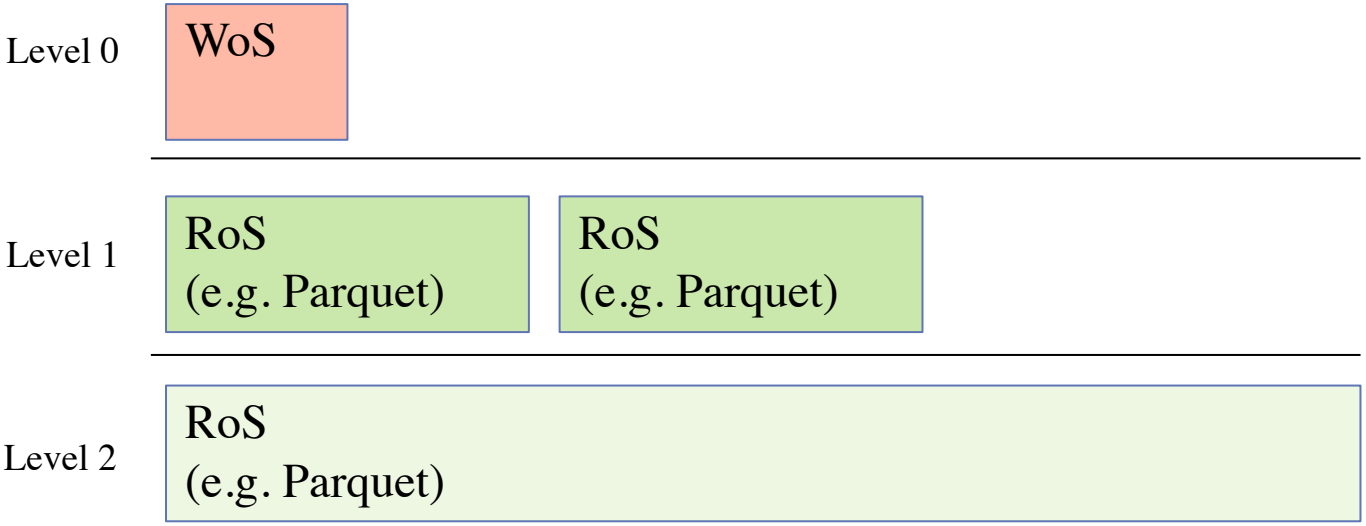
Principle:

- Append updates to a write-optimized store (WoS) or logfile
- Regularly merge WoS with RoS (read-optimized store)



Multi-level Storage Compaction

- Append updates to a write-optimized store (WoS) or logfile
- Regularly merge WoS with RoS (read-optimized store)
  - Popular idea: Log-Structured Merges (~= “Compaction”)
    - organize data in layers, each layer a factor **X** bigger than previous
    - Top-layer is WoS, other layers are columnar compressed
    - When a layer exceeds layer max-size: merge it into the next layer, and empty it
  - Log structured merging has log**X**(N) levels
    - Amortized updates have O(log(N)) cost!



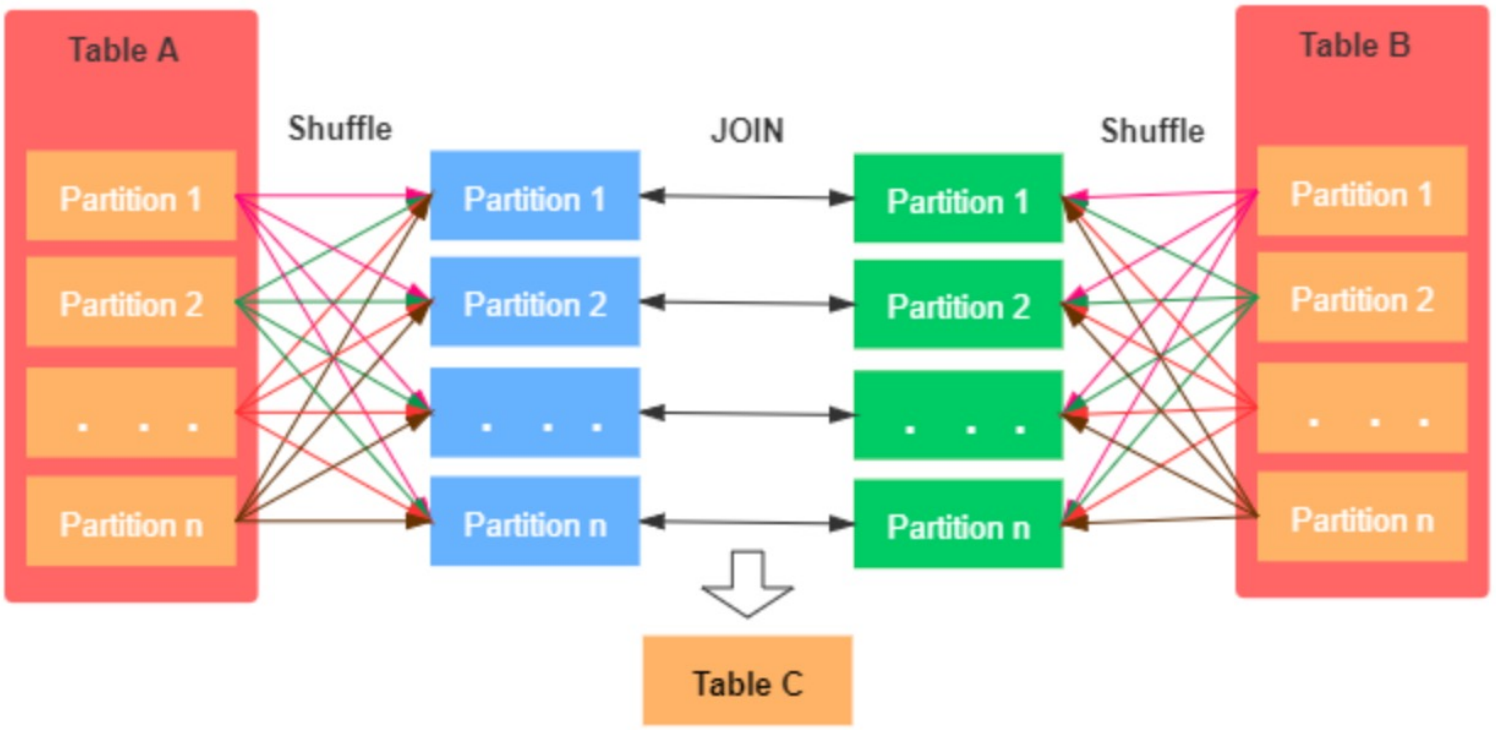
Summary

- Big Data seemed apart from classical databases in the Hadoop
  - but SQL systems for Big Data have incorporated many DB ideas
  - e.g. columnar storage, compression, vectorized execution, JIT
  - …and more (did not discuss query optimization, parallellism)
- Additional, SQL for Big Data systems have to deal with
  - Cluster computing (data distribution, data locality)
  - Cloud computing (storage services, ephemeral storage)

This lecture aimed at giving you insight in some of the important “under the hood” aspects of large-scale data systems.

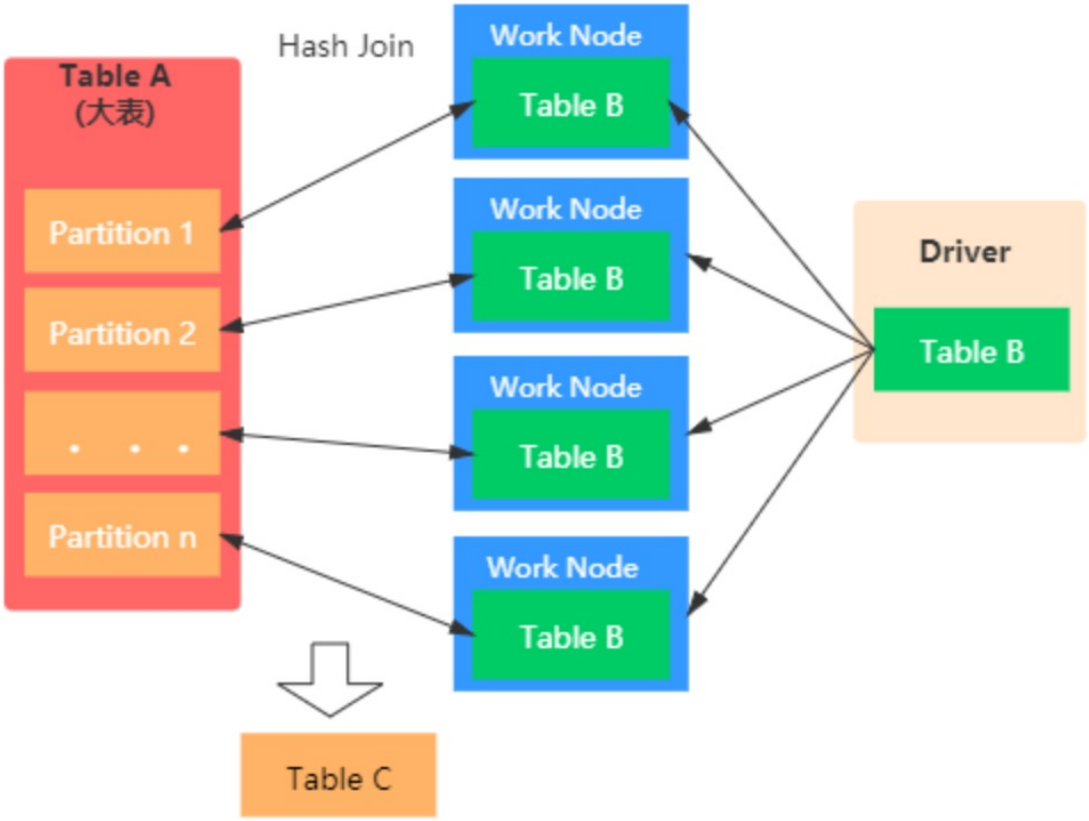
Shuffle-based Join

- This type of join has very significant communication
  - +serialization/deserialization
- Re-partitioning triggers full materialization in Spark blocks (cache+disk)



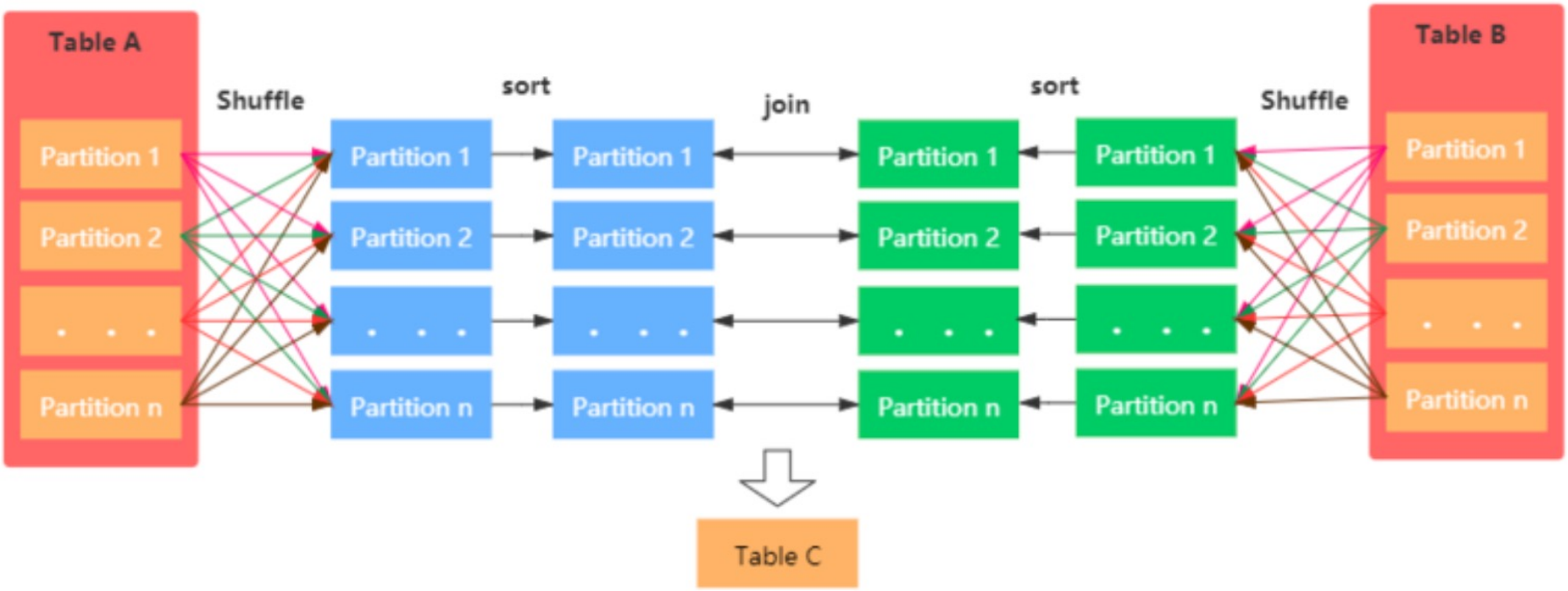
Broadcast Join

- Note that the driver node is *heavily* involved and can be overwhelmed



Sort-merge Join

- The default join method in Spark!
- Same or higher cost than broadcast hash join



How Spark Chooses the Join

if there are join hints:

- Broadcast hint: select broadcast hash join if the join type supports it
- Sort merge hint: select sort merge join
- Shuffle hash hint: if the join type supports it, select shuffle hash join
- Shuffle replicate NL hint: if it is inner join, select Cartesian product

If there are no join hints:

- If the join type is supported and one of the tables can be broadcast（**spark.sql.autoBroadcastJoinThreshold**, the default is 10MB), then select broadcast hash join
- If the parameter **spark.sql.join.prefersortmergejoin is set to false**, and a table is small enough (you can build a hash map), select shuffle hash join
- If the join keys are sorted, select sort merge join (often unknown to spark)
- select Cartesian join

Ideas for Reorg

- Writing parquet files
- Throwing away non-local knows
- Throwing away friendless persons
- Throwing away person-less interests
- Interests: Inverted Files on Artists
  - Order interests on Artist ➔ rely on MinMax index
  - Partition into multiple files?
    - On artist id
- Storing persons and knows together
  - Nested Parquet files
- Birthdate Locality
  - Rely on MinMax index
- Denormalization
  - Precompute the triangles

Ideas for Cruncher