CMSC 724: Database Management Systems
Storage

Instructor: Amol Deshpande
amol@cs.umd.edu

Outline

- Basics
- PAX: Within-page Columnar Storage
- Compression in Column-Stores
- Dremel: Storing Hierarchical Data
- Delta Lake: Storage Issues in Data Lakes

Data Storage Options

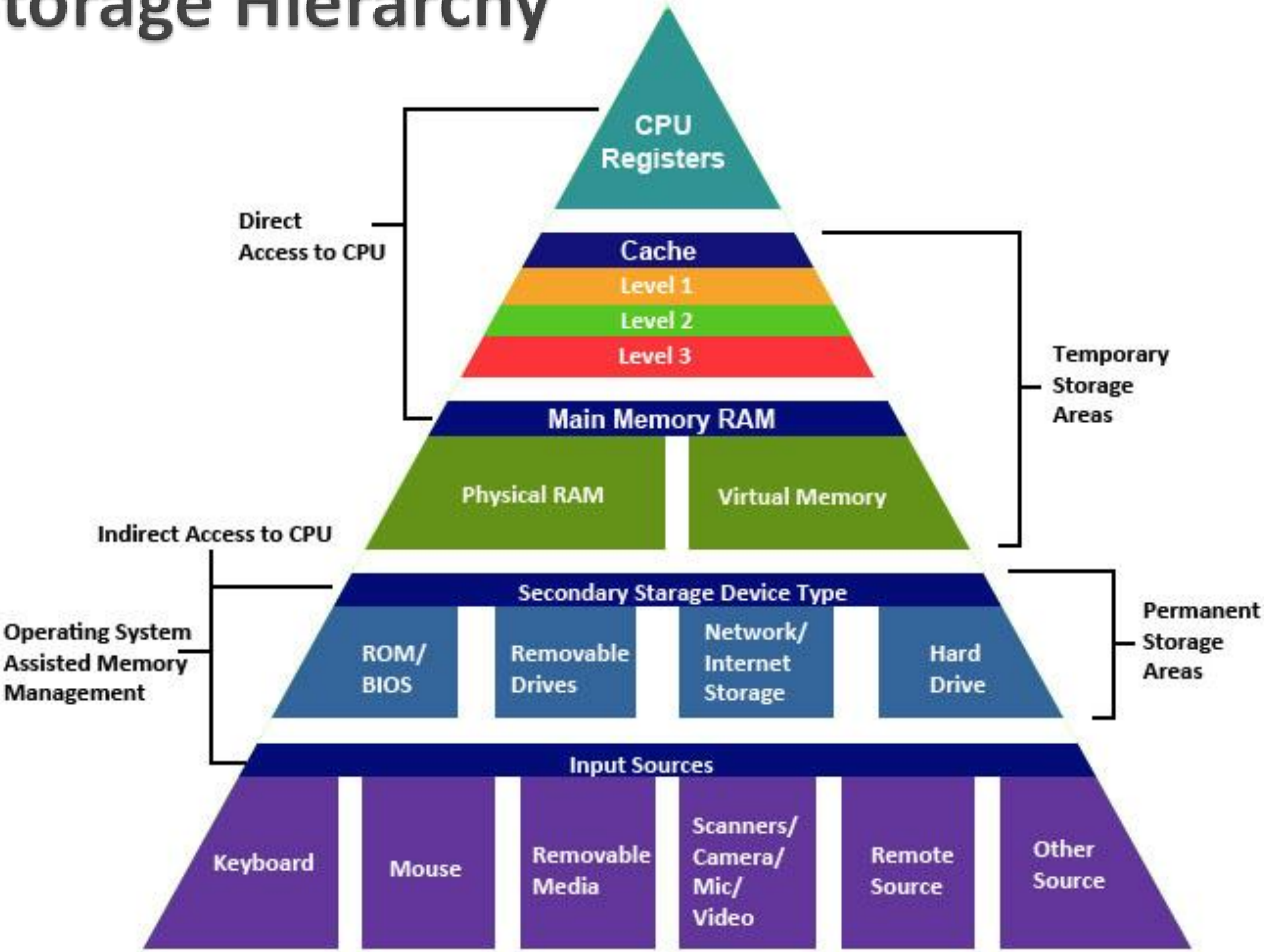
- At various points, data stored in different storage hardware
  - Memory, Disks, SSDs, Tapes, Cache
  - Tradeoffs between speed and cost of access
  - CPU needs the data in memory and cache to operate on it

- Volatile vs nonvolatile
  - Volatile: Loses contents when power switched off

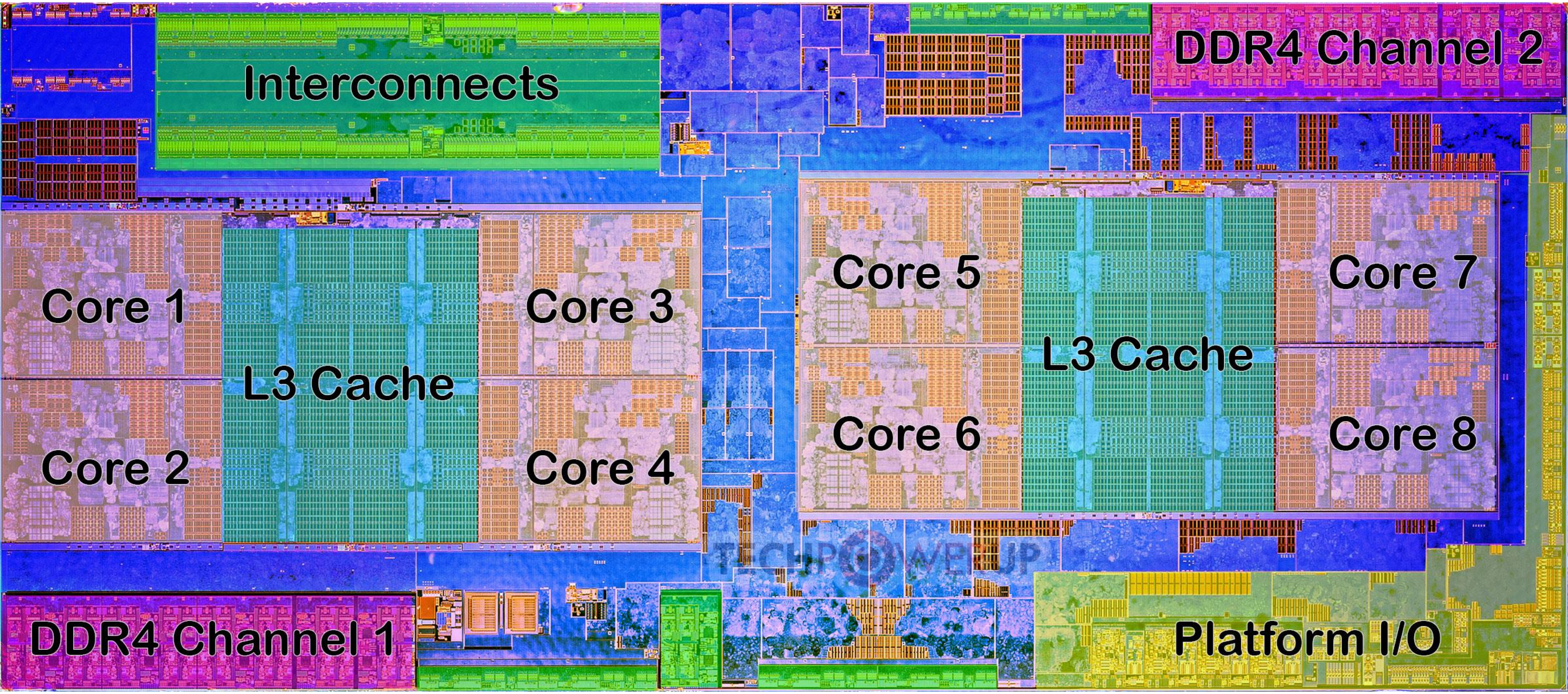
- Sequential vs random access
  - Sequential: read the data contiguously
    - select * from employee
  - Random: read the data from anywhere at any time
    - select * from employee where name like '__a__b'

source: https://cse1.net/recaps/4-memory.html

Storage Hierarchy



AMD Ryzen CPU Architecture

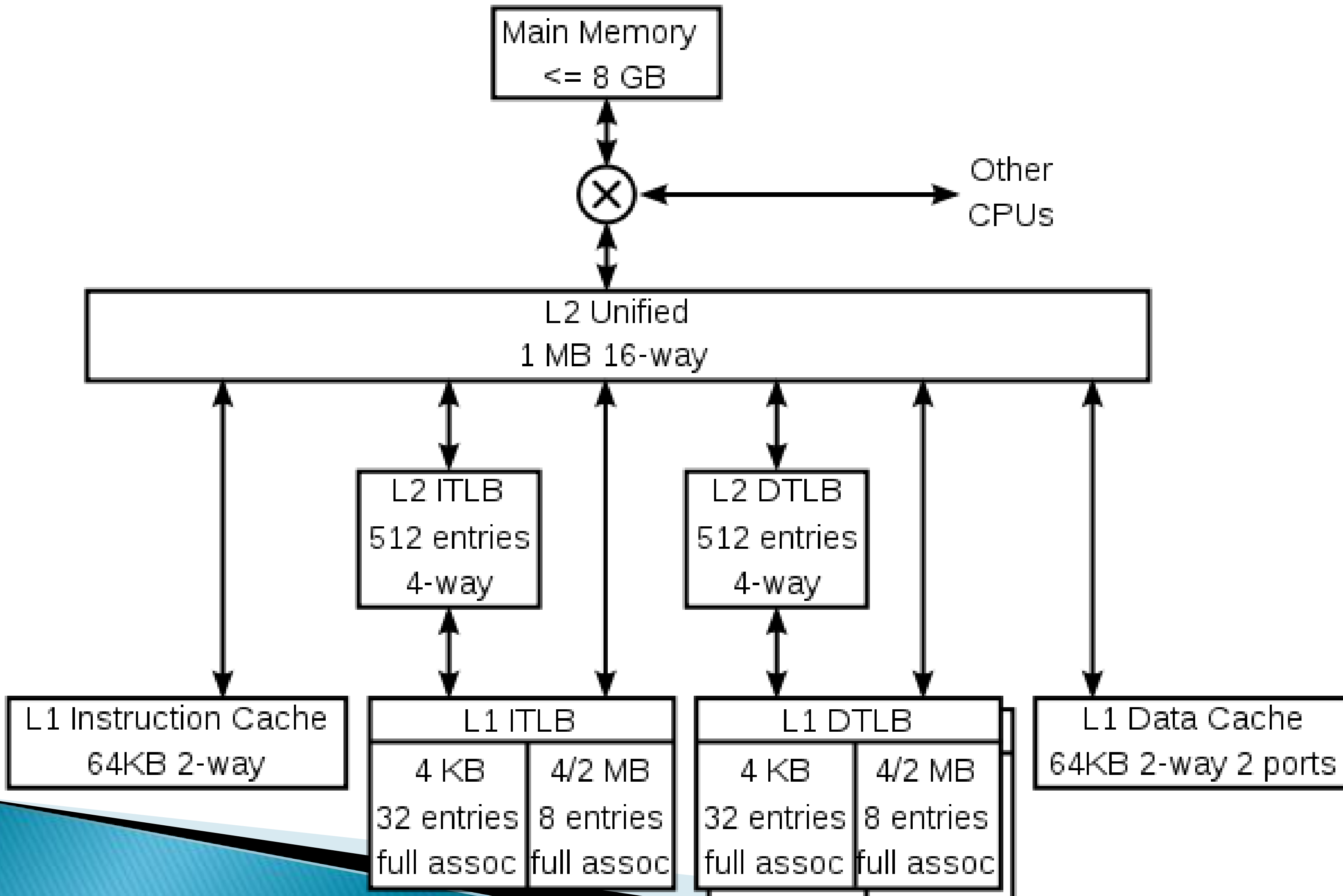


Die shot overlaid with functional units

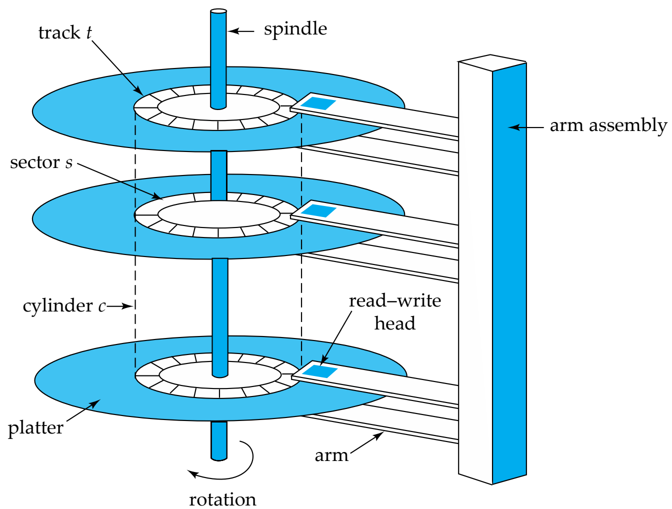
https://www.techpowerup.com/review/amd-ryzen-5-1600/3.html

Storage Hierarchy: Cache

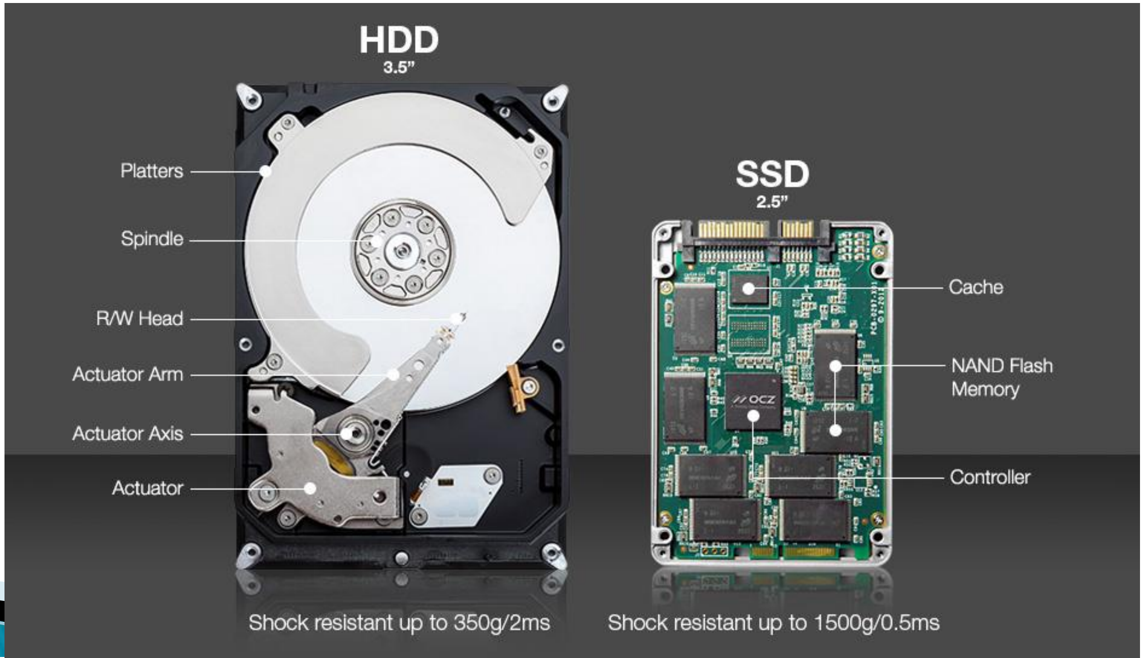
K8 core in the AMD Athlon 64 CPU



Disks vs SSDs



1956
IBM RAMAC



Data Storage Options

- Hard disks dominant form of storage for a long time
  - About 10ms per random access ➔ at most 100 random reads per second
  - vs up to 500 MB/s sequential I/O

- Many traditional database design decisions driven by:
  - Huge volumes of data on disks + Low amounts of memory + Low-speed networks
  - ➔ Communication between disks and memory the main bottleneck

- Solid state drives much more common today
  - No seeks → Much better random reads
  - Writes require erasing an entire block, and rewriting it
  - SSDs provide a similar interface of "blocks"

Shifting Tradeoffs

- Much faster networks
- Often cheaper to access another computer's memory than accessing your own disk (in data centers)
- Cache is playing more and more important role
- Data often fits in memory of a single machine, or a cluster of machines
- "Disk" considerations less important
  - Still: Disks are where most of the data lives today

Mapping Tuples to Disk Blocks

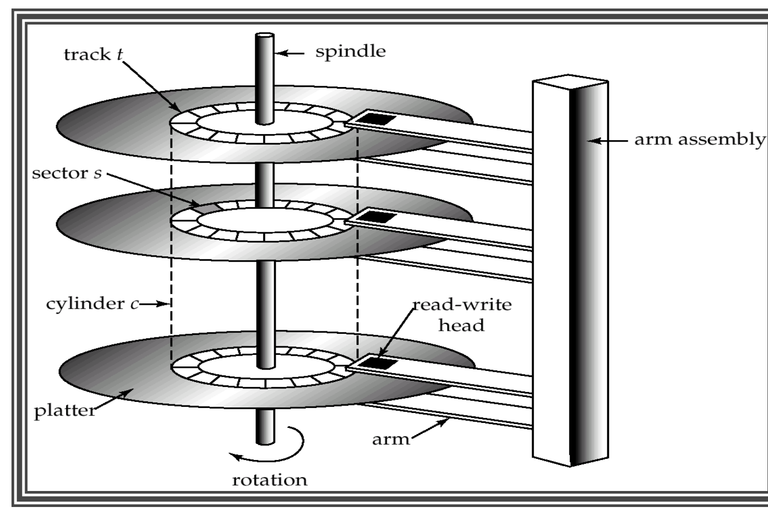
ID	*name*	*salary*	*dept_name*	*building*	*budget*
22222	Einstein	95000	Physics	Watson	70000
12121	Wu	90000	Finance	Painter	120000
32343	El Said	60000	History	Painter	50000
45565	Katz	75000	Comp. Sci.	Taylor	100000
98345	Kim	80000	Elec. Eng.	Taylor	85000
76766	Crick	72000	Biology	Watson	90000
10101	Srinivasan	65000	Comp. Sci.	Taylor	100000
58583	Califieri	62000	History	Painter	50000
838[illegible]	[illegible]	[illegible]	[illegible]	Taylor	100000
15151	Mozart	40000	Music	Packard	80000
33456	Gold	87000	Physics	Watson	70000
76543	Singh	80000	Finance	Painter	120000

ID	*name*	*dept_name*	*salary*
22222	Einstein	Physics	95000
12121	Wu	Finance	90000
32343	El Said	History	60000
45565	Katz	Comp. Sci.	75000
98345	Kim	Elec. Eng.	80000
76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
58583	Califieri	History	62000
83821	Brandt	Comp. Sci.	92000
15151	Mozart	Music	40000
33456	Gold	[illegible]	[illegible]
76543	Singh	Finance	80000

dept_name	*building*	*budget*
Comp. Sci.	Taylor	100000
Biology	Watson	90000
Elec. Eng.	Taylor	85000
Music	Packard	80000
Finance	Painter	120000
History	Painter	50000
Physics	Watson	70000

- Very important implications on performance
- Quite a few different ways to do this
- Similar issues even if not using disks as the primary storage

?



File System or Not

- Option 1: Use OS File System
  - File systems are a standard abstraction provided by Operating Systems (OS) for managing data
  - Major Con: Databases don't have as much control over the physical placement anymore --- OS controls that
    - E.g., Say DBMS maps a relation to a "file"
    - No guarantee that the file will be "contiguous" on the disk
    - OS may spread it across the disk, and won't even tell the DBMS

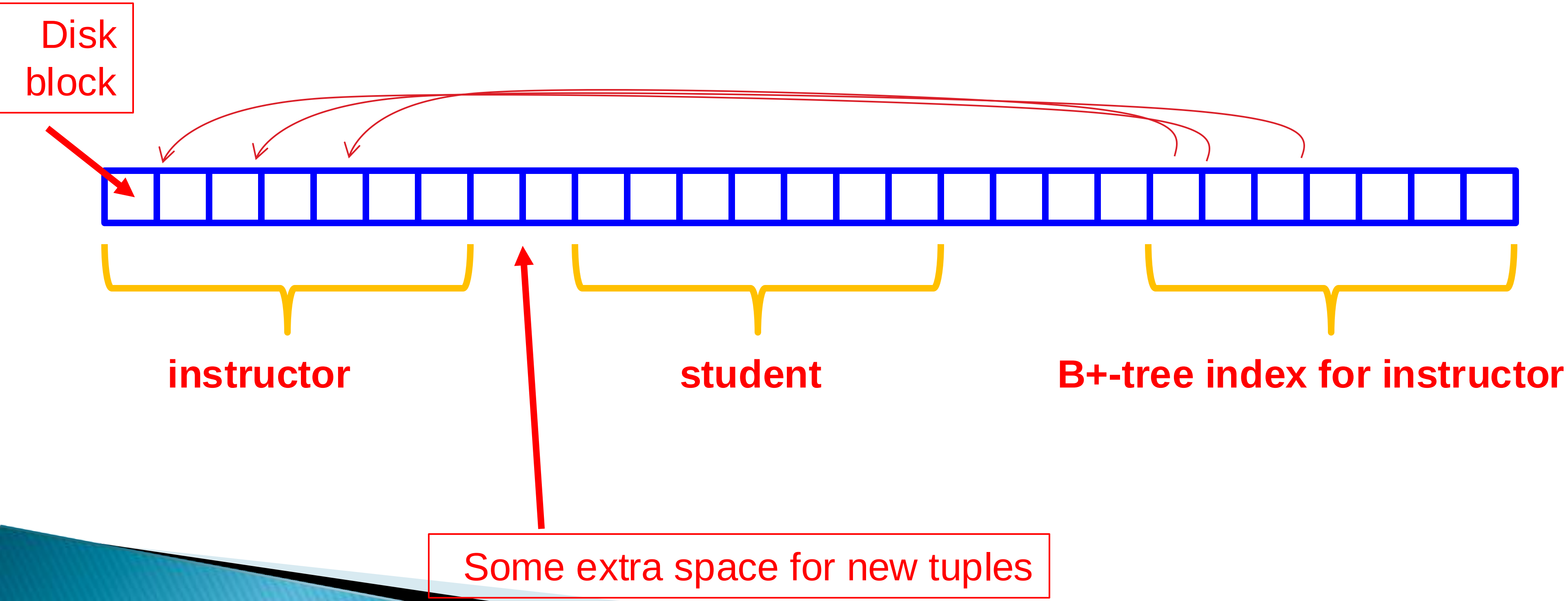
- Option 2: DBMS directly works with the disk or uses a lightweight/custom OS
  - Increasingly uncommon – most DBMSs today run on top of OSes (e.g., PostgreSQL on your laptop, or on linux VMs in the cloud, or on a distributed HDFS)

Through a File System

- Option 1: Allocate a single “file” on the disk, and treat it as a contiguous sequence of blocks
  - This is what PostgreSQL does
  - The blocks may not actually be contiguous on disk
- Option 2: A different file per relation
  - Some of the simpler DBMS use this approach
- Either way: we have a set of relations mapped to a set of blocks on disk

Example

- Each relation stored separately on a separate set of blocks
  - Assumed to be contiguous
- Each “index” maintained in a separate set of blocks
  - Assumed to be contiguous

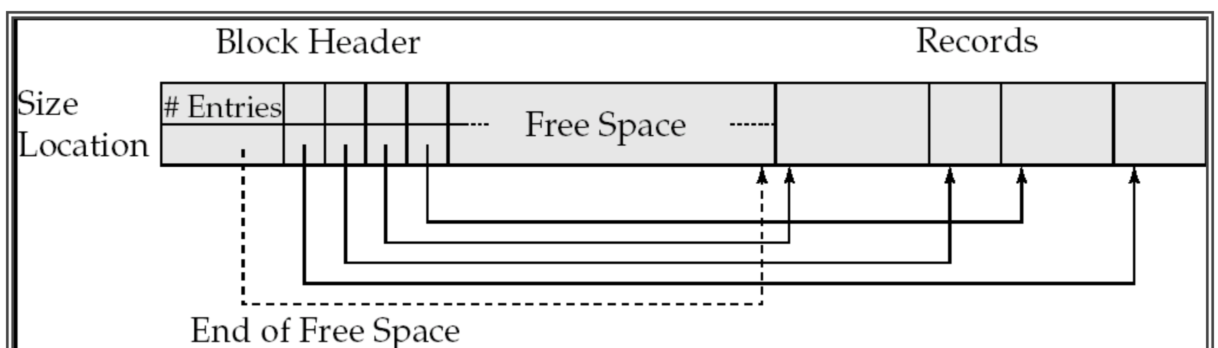


Within a Single Block: NSM Model

header				
record 0	10101	Srinivasan	Comp. Sci.	65000
record 1				
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4				
record 5	33456	Gold	Physics	87000
record 6				
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

Fixed Length Records

Slotted page/block structure



Decomposition Storage Model (DSM)

- Store the data column-wise
- Need to maintain an “index” with each value (to be able to stitch them together)
- Good for queries that read few columns, but bad for writes or queries that read all columns

R1

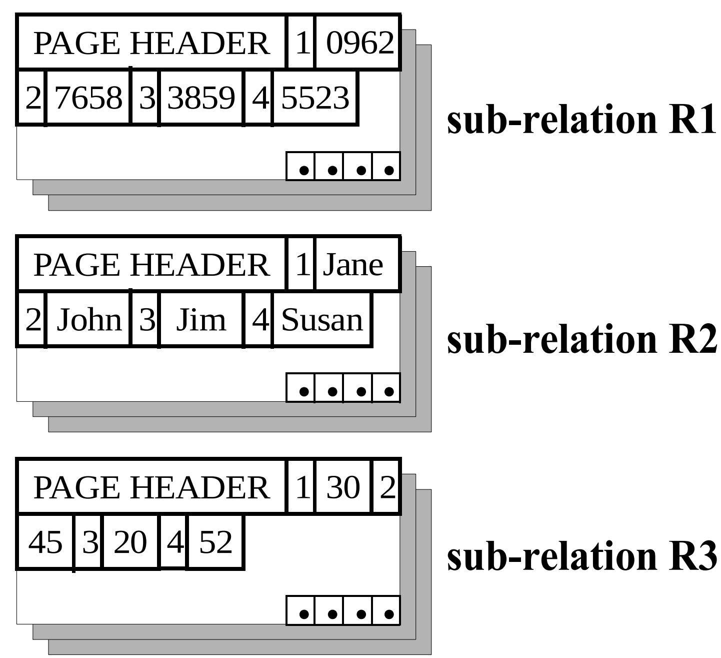
RID	SSN
1	0962
2	7658
3	3859
4	5523
5	9743
6	0618

R2

RID	Name
1	Jane
2	John
3	Jim
4	Susan
5	Leon
6	Dan

R3

RID	Age
1	30
2	45
3	20
4	52
5	43
6	37

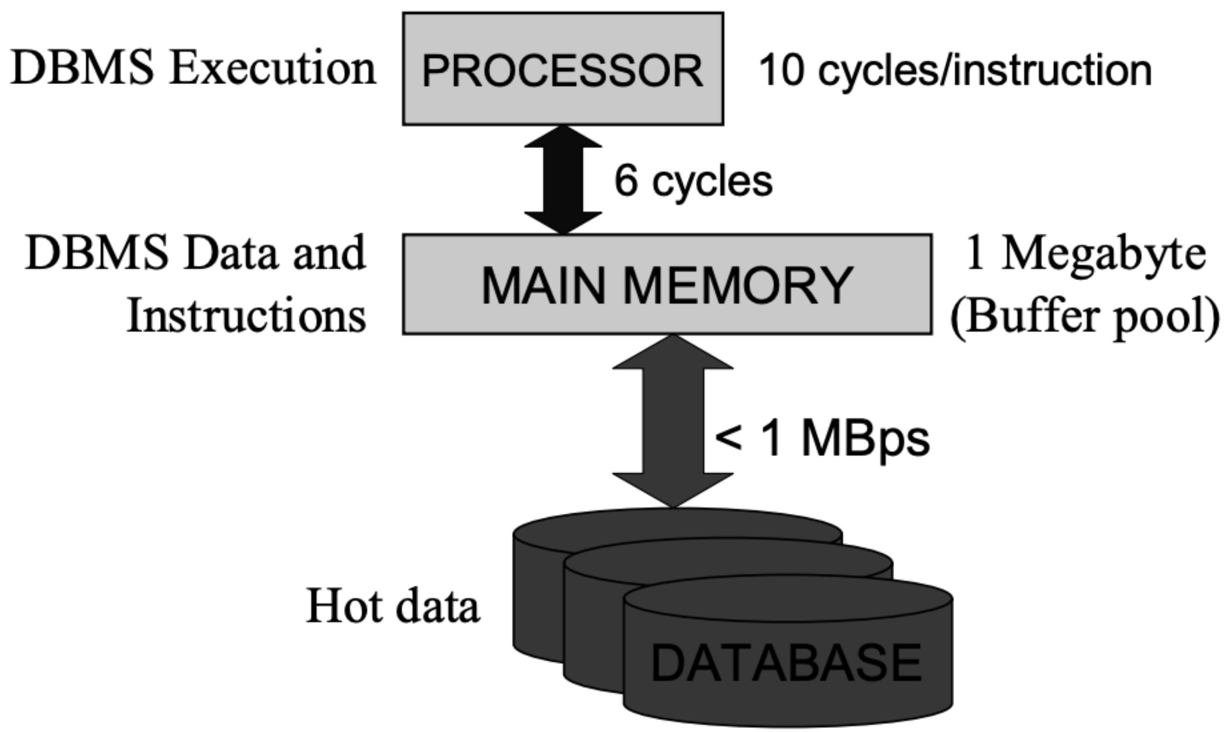


Outline

- Basics
- PAX: Within-page Columnar Storage
- Compression in Column-Stores
- Dremel: Storing Hierarchical Data
- Delta Lake: Storage Issues in Data Lakes

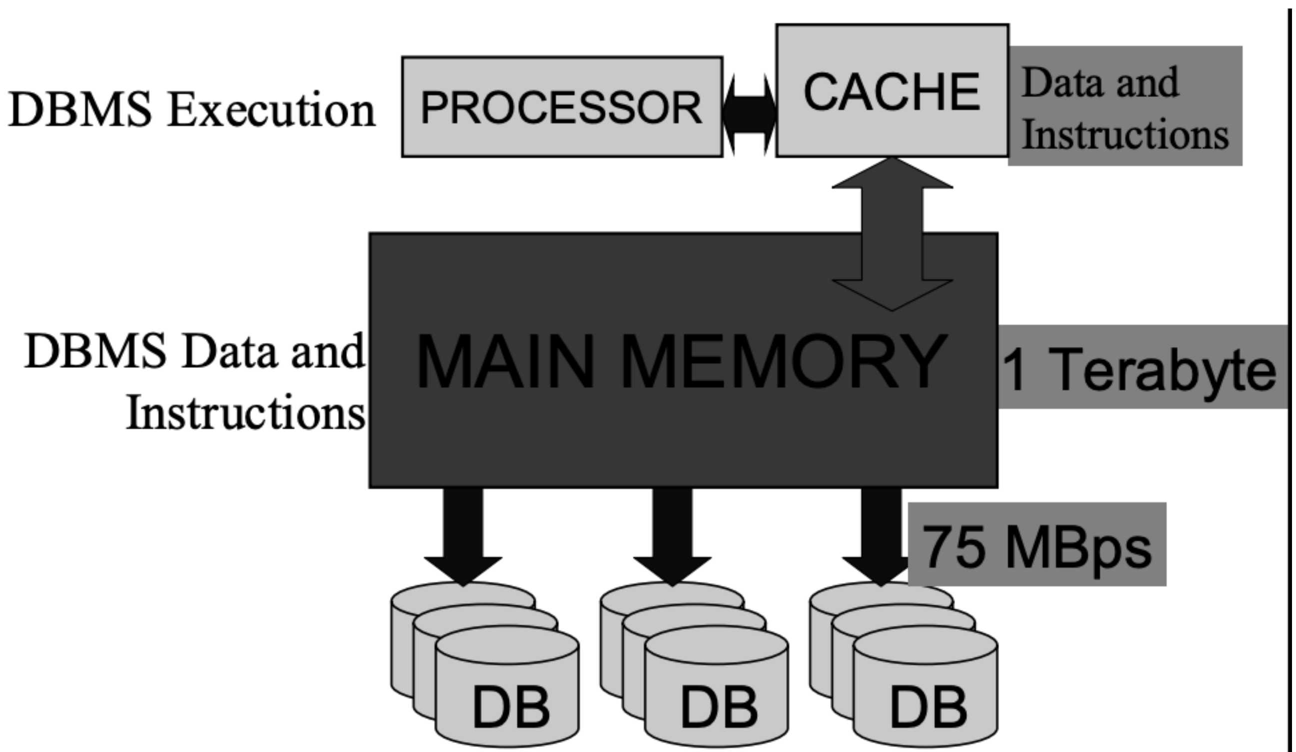
Shifting Tradeoffs

- Computer platforms in 1980: main performance bottleneck I/O Latency



Shifting Tradeoffs

- Computer platforms today: hot data migrates to larger and slower main memory – almost no disk I/Os



Shifting Tradeoffs

- DBMSs on a Modern Processor: Where does time go? VLDB 1999
- Detailed profiling on a few simple queries on different databases

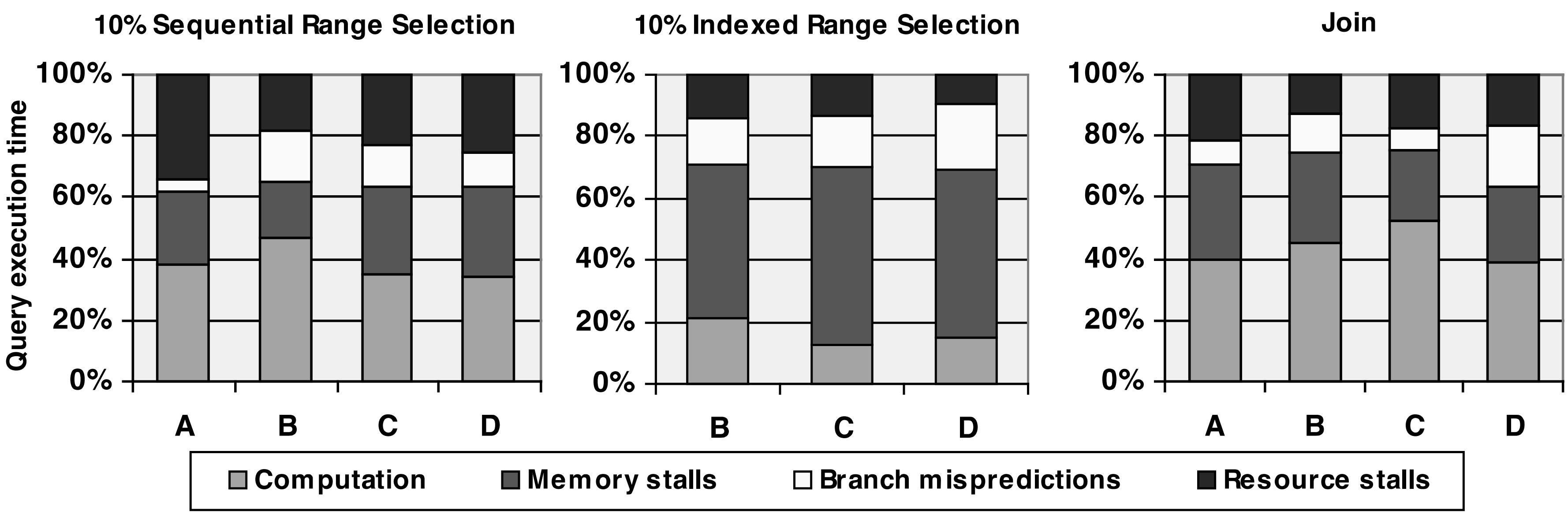


Figure 5.1: *Query execution time breakdown into the four time components.*

Shifting Tradeoffs

- DBMSs on a Modern Processor: Where does time go? VLDB 1999
- Detailed profiling on a few simple queries on different databases

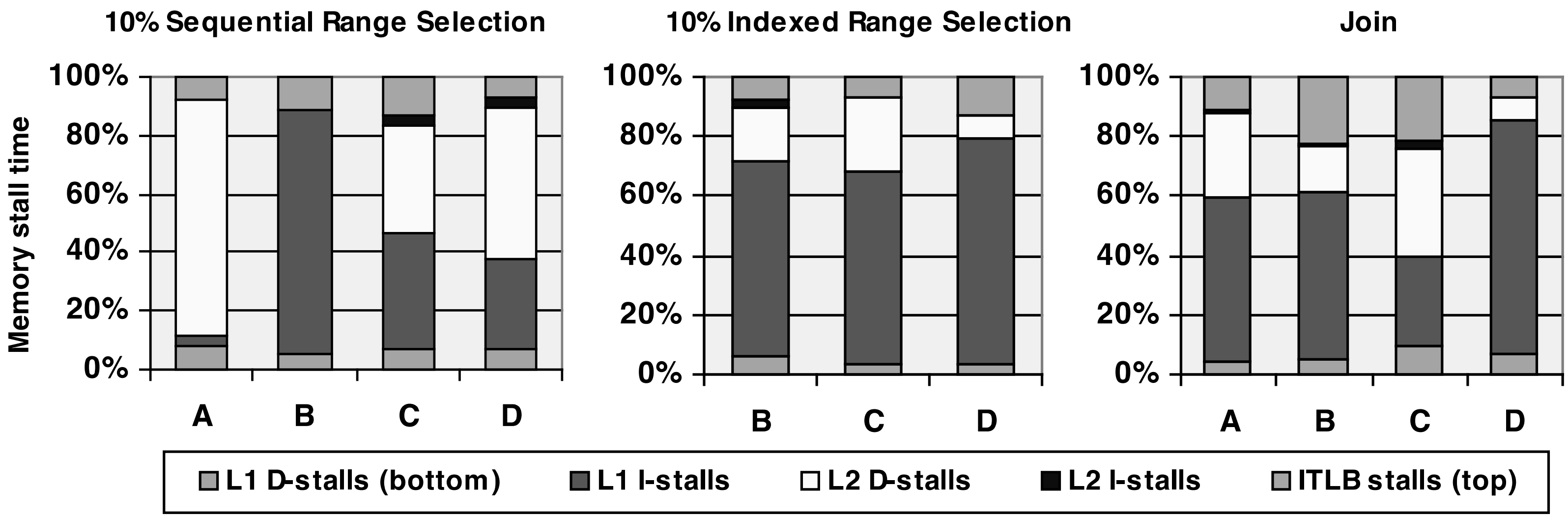
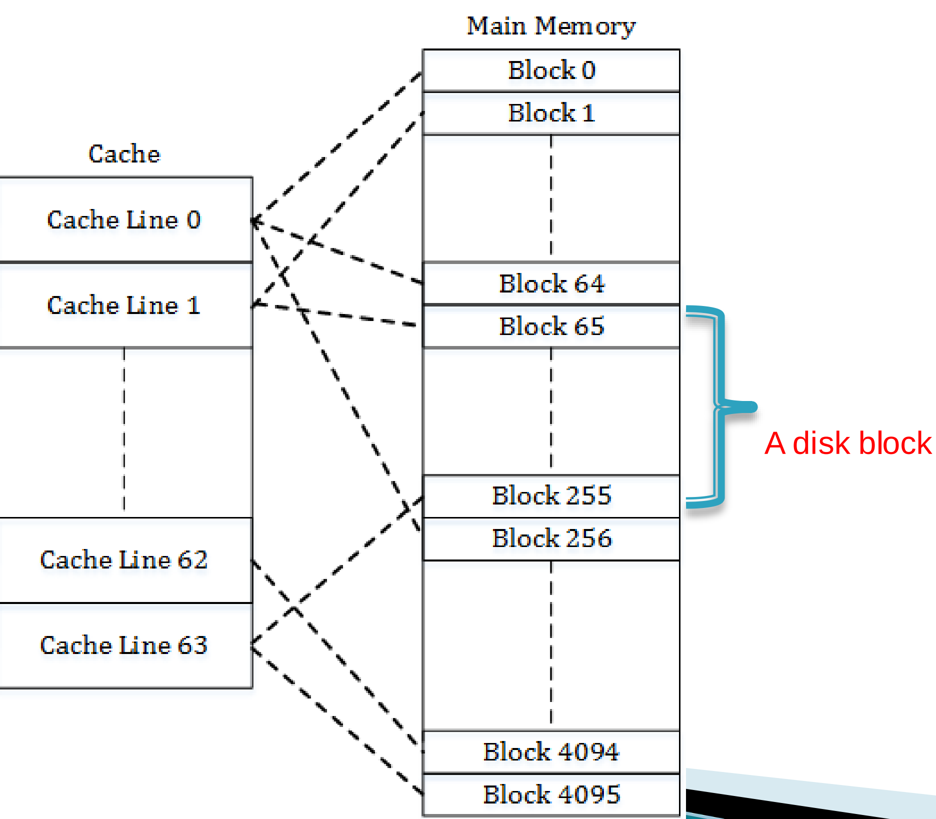


Figure 5.2: *Contributions of the five memory components to the memory stall time (T_M)*

PAX: Motivation

- Cache misses are a major source of delays in modern systems
- Only a fraction of the data transferred to the cache is useful
  - Typical cache line sizes: 64-256 bytes



Memory divided into ”blocks” == size of the cache line

”Disk blocks” often memory-mapped (or loaded directly into memory)

If using n-ary storage, unnecessary attributes of a tuple are loaded into the cache

PAX: Motivation

- Cache misses are a major source of delays in modern systems
- Only a fraction of the data transferred to the cache is useful
  - Typical cache line sizes: 64-256 bytes

select *name*
from *R*
where *age* < 40;

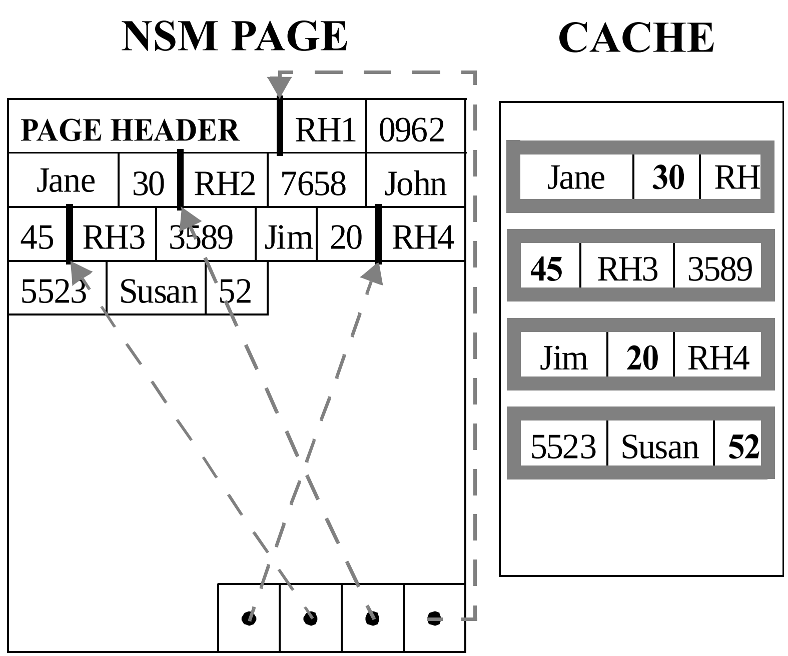


FIGURE 1: *The cache behavior of NSM.*

PAX

- Combine the best properties of the two models

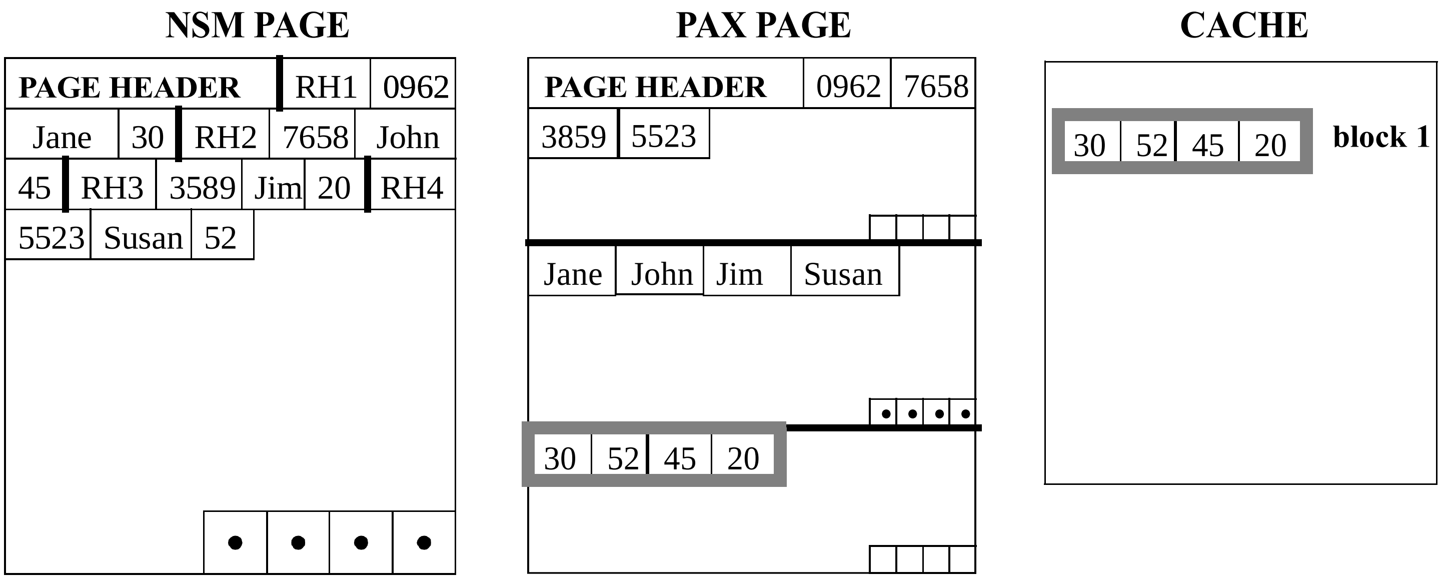


FIGURE 3: Partition Attributes Across (PAX), and its cache behavior. *PAX partitions records into minipages within each page. As we scan R to read attribute* age*, values are much more efficiently mapped onto cache blocks, and the cache space is now fully utilized.*

PAX: Implementation in Shore

- Page sub-divided into minipages (with flexible boundaries)
- Boundaries dynamically adjusted based on the sizes of the variable-length fields
- During bulk-loading, use the boundaries from the previous page as a starting point
- Updates may cause re-org
- Deletions handled by marking deleted records and reusing during inserts

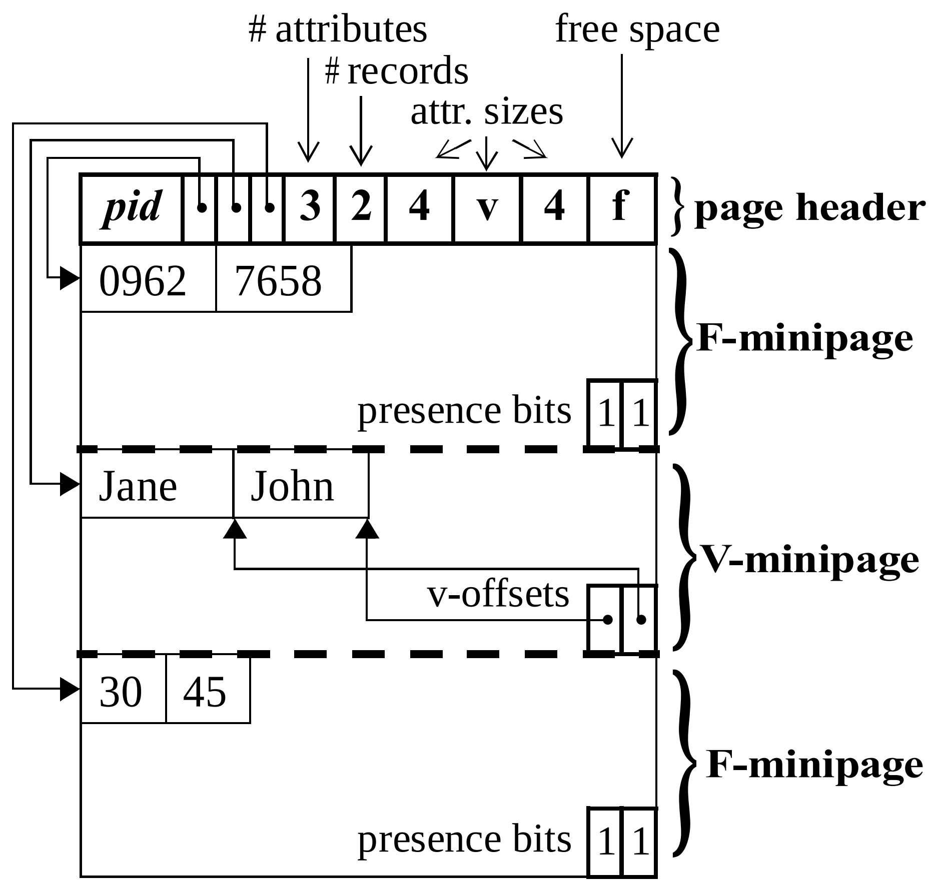


FIGURE 4: *An example PAX page.*

PAX: Implementation in Shore

- Scans in Shore Implementation
  - NSM: One scan operator per relation
  - PAX or DSM: One scan operator per attribute being read
    - So the reading of the different attributes (for the same record) going on in parallel

- Standard hybrid hash join algorithm
  - Build hash partitions on left relation, and probe using the right relation
  - Uses scan operators to read and construct the tuples

Experimental Results

select avg(a_p**)**
from R
where a_q > Lo and a_q < Hi

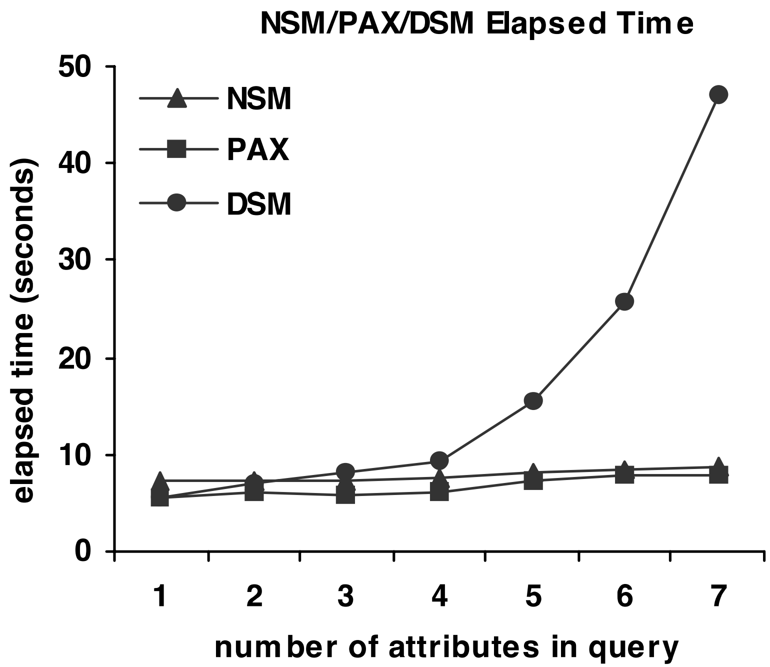


FIGURE 6: *Elapsed time comparison as a function of the number of attributes in the query.*

Experimental Results

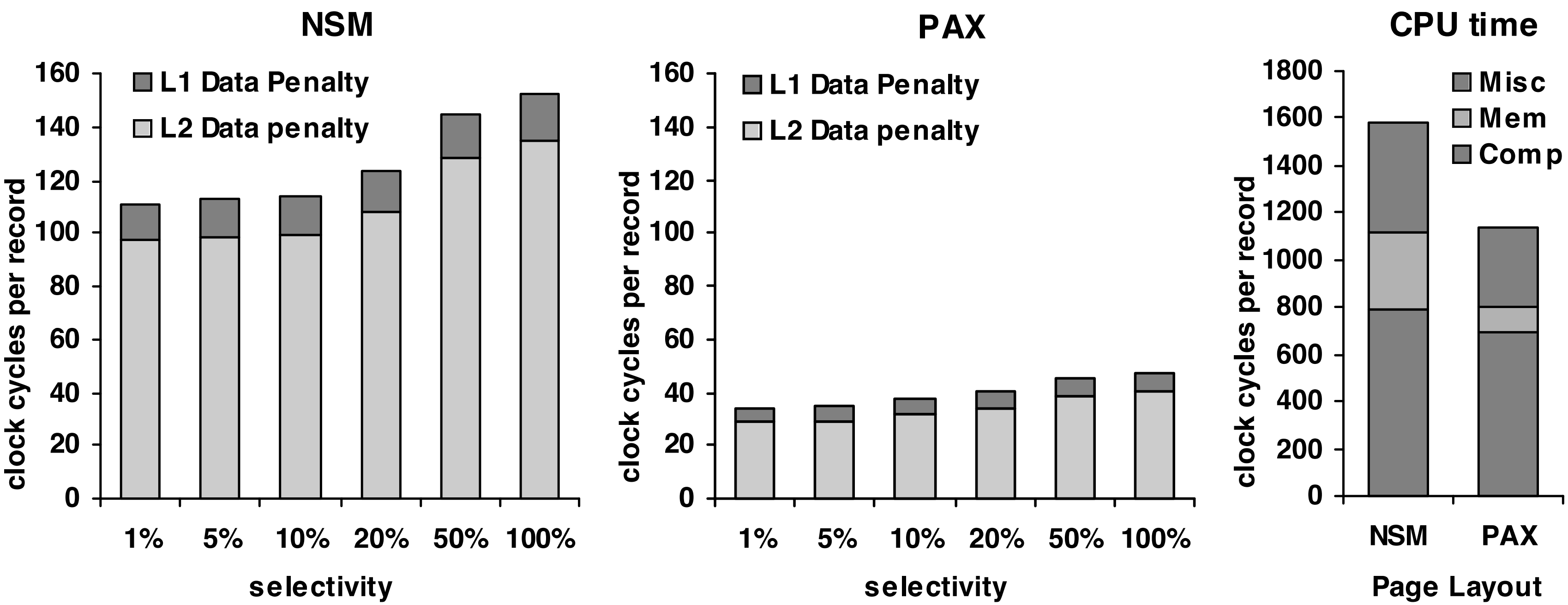


FIGURE 7: *PAX impact on memory stalls*

Experimental Results

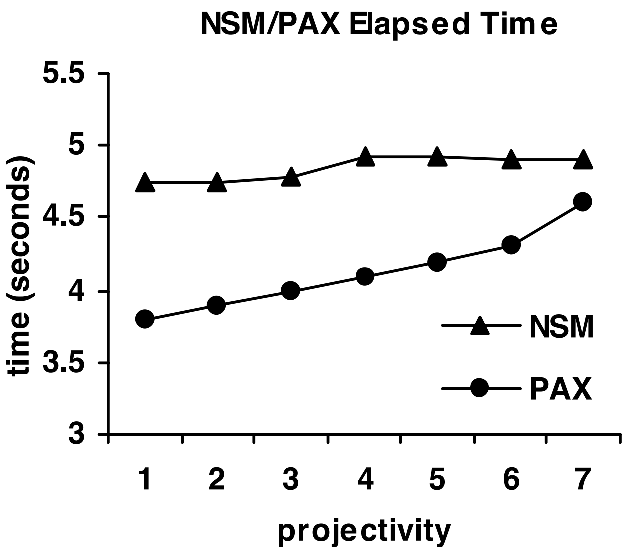


FIGURE 8: *PAX/NSM sensitivity to projectivity.*

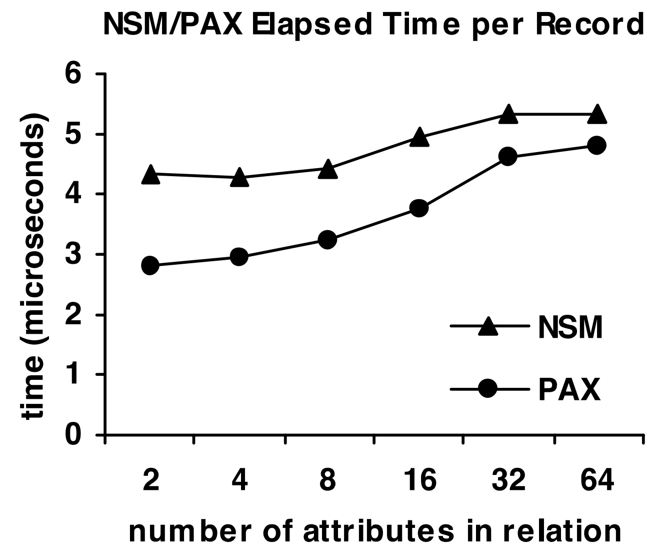


FIGURE 9: *PAX/NSM sensitivity to the number of attributes in the relation.*

Experimental Results

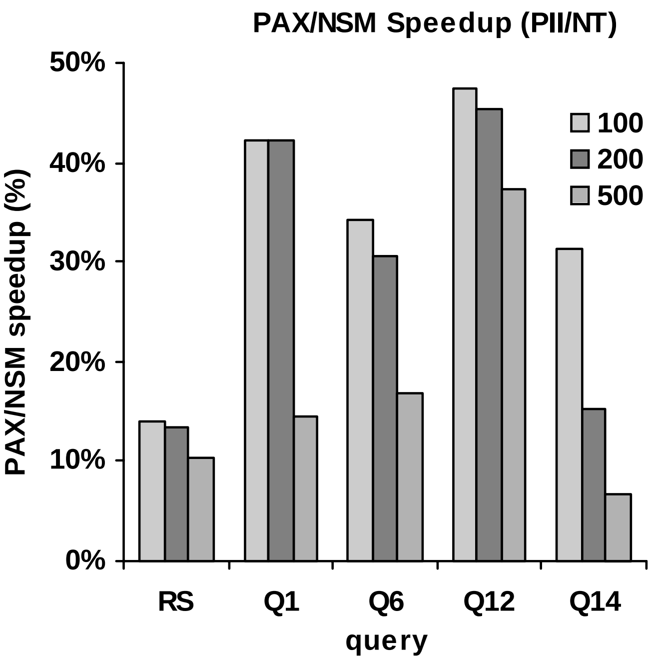


FIGURE 11: *PAX/NSM speedup on read-only queries.*

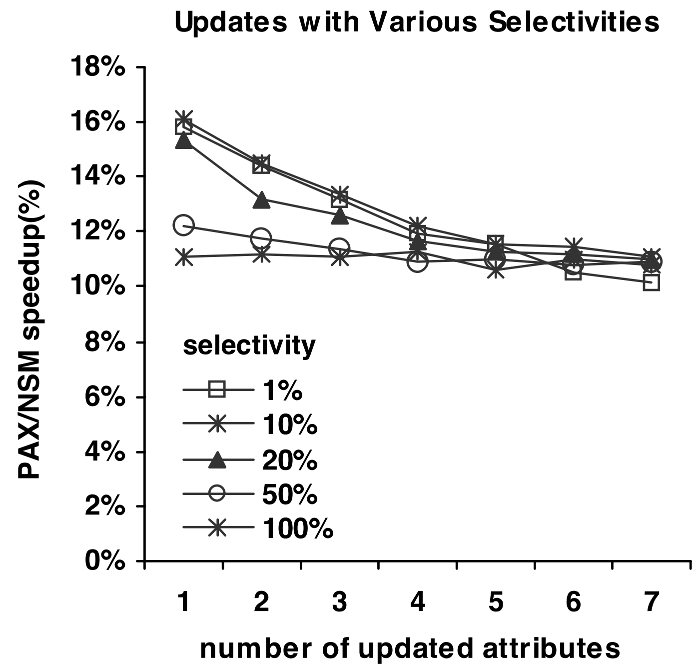
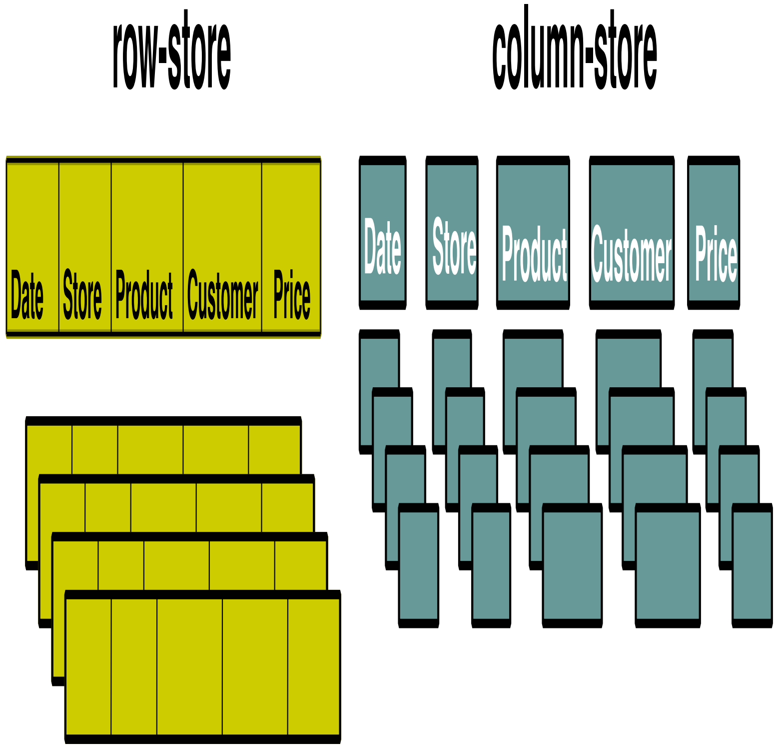


FIGURE 12: *PAX/NSM speedup on updates.*

Outline

- Basics
- PAX: Within-page Columnar Storage
- Compression in Column-Stores
- Dremel: Storing Hierarchical Data
- Delta Lake: Storage Issues in Data Lakes

Column-stores vs Row-stores



row-store	column-store
+ easy to add/modify a record	+ only need to read in relevant data
- might read in unnecessary data	- tuple writes require multiple accesses

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

Material from Tutorial on Column-stores by Harizopoulos, Abadi, Boncz

One size fits all? Part 2: Benchmarking

- Using a "Telco" schema
- "usage" table has 200 columns: only 7 columns need to be used
- Much higher compression ratios (factor of 10 vs 3)
- Row store (in comparison) didn't use indexing

```
SELECT account.account_number,
        sum (usage.toll_airtime),
        sum (usage.toll_price)
FROM    usage, toll, source, account
WHERE   usage.toll_id = toll.toll_id
  AND   usage.source_id = source.source_id
  AND   usage.account_id = account.account_id
  AND   toll.type_ind in ('AE'. 'AA')
  AND   usage.toll_price > 0
  AND   source.type != 'CIBER'
  AND   toll.rating_method = 'IS'
  AND   usage.invoice_date = 20051013
GROUP BY  account.account_number
```

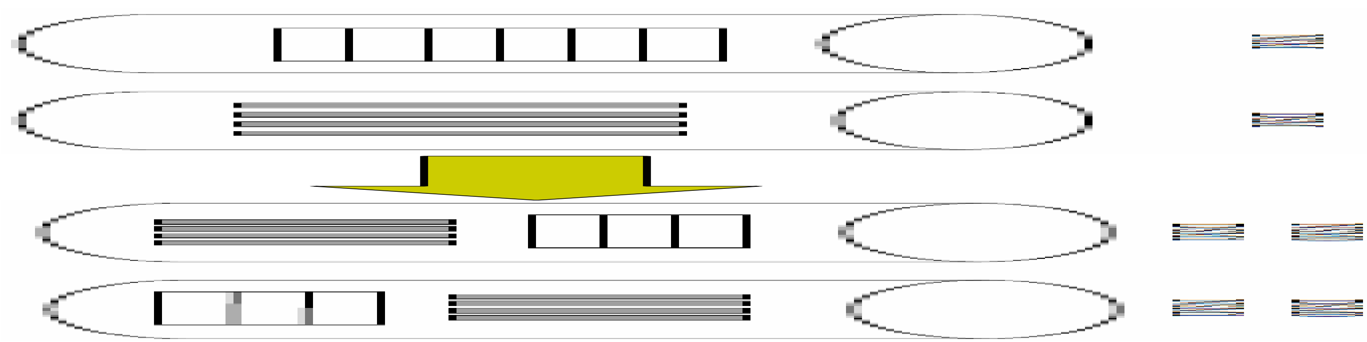
Figure 3. Query 2

	Vertica	*Appliance*
Query 1	2.06	300
Query 2	2.20	300
Query 3	0.09	300
Query 4	5.24	300
Query 5	2.88	300

Figure 2. Query Running Times (seconds)

Fractured Mirrors

- Ramamurthy, DeWitt, Su; VLDB 2002
- Mirroring often used for fault tolerance
- Store the mirrors in different format



C-Store

- Relational model + SQL on top, but a complete redesign of storage and query execution
- All data stored in the form of "projections"
  - In essence: "materialized views"

Name	Age	Dept	Salary
Bob	25	Math	10K
Bill	27	EECS	50K
Jill	24	Biology	80K

Table 1: Sample EMP data

Dname	Floor
Math	1
EECS	2

DEPT Table

```
EMP1(name, age| age)
EMP2(dept, age, DEPT.floor| DEPT.floor)
EMP3(name, salary| salary)
DEPT1(dname, floor| floor)
```

Example 2: Projections in Example 1 with sort orders

Jill	24
Bob	25
Bill	27

EMP1

Name	Age
Jill	24
Bob	25
Bill	27

C-Store

- Relational model + SQL on top, but a complete redesign of storage and query execution
- All data stored in the form of "projections"
  - In essence: "materialized views"

Name	Age	Dept	Salary
Bob	25	Math	10K
Bill	27	EECS	50K
Jill	24	Biology	80K
Sara	29	Math	90K

Table 1: Sample EMP data

```
EMP1(name, age| age)
EMP2(dept, age, DEPT.floor| DEPT.floor)
EMP3(name, salary| salary)
DEPT1(dname, floor| floor)
```

Example 2: Projections in Example 1 with sort orders

Dname	Floor
Math	1
EECS	2
Biology	4

DEPT Table

dept	age	dept.floor
Math	25	1
Math	29	1
EECS	27	2
Biology	24	4

EMP2

C-Store

- Need some way to reconstruct the tuples
- Use “join indexes”
- Each column in multiple different projections to reduce the need to do joins
  - Ideally each query covered by a single projection

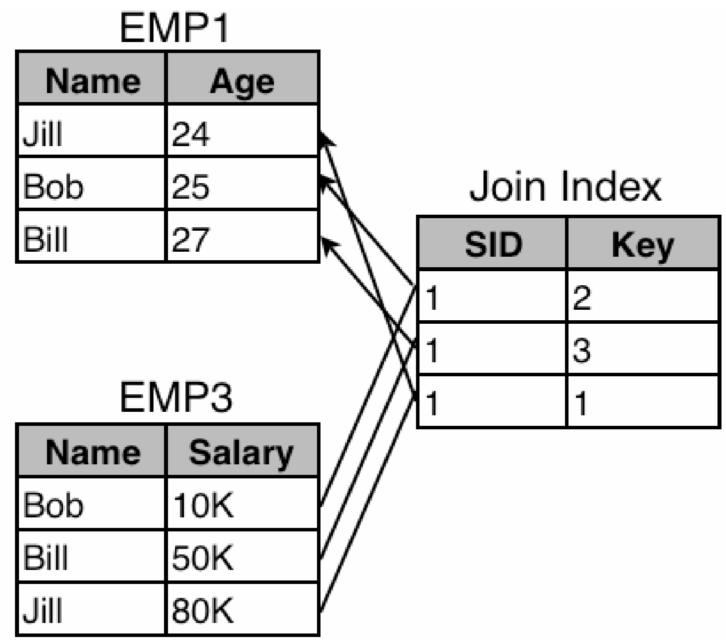
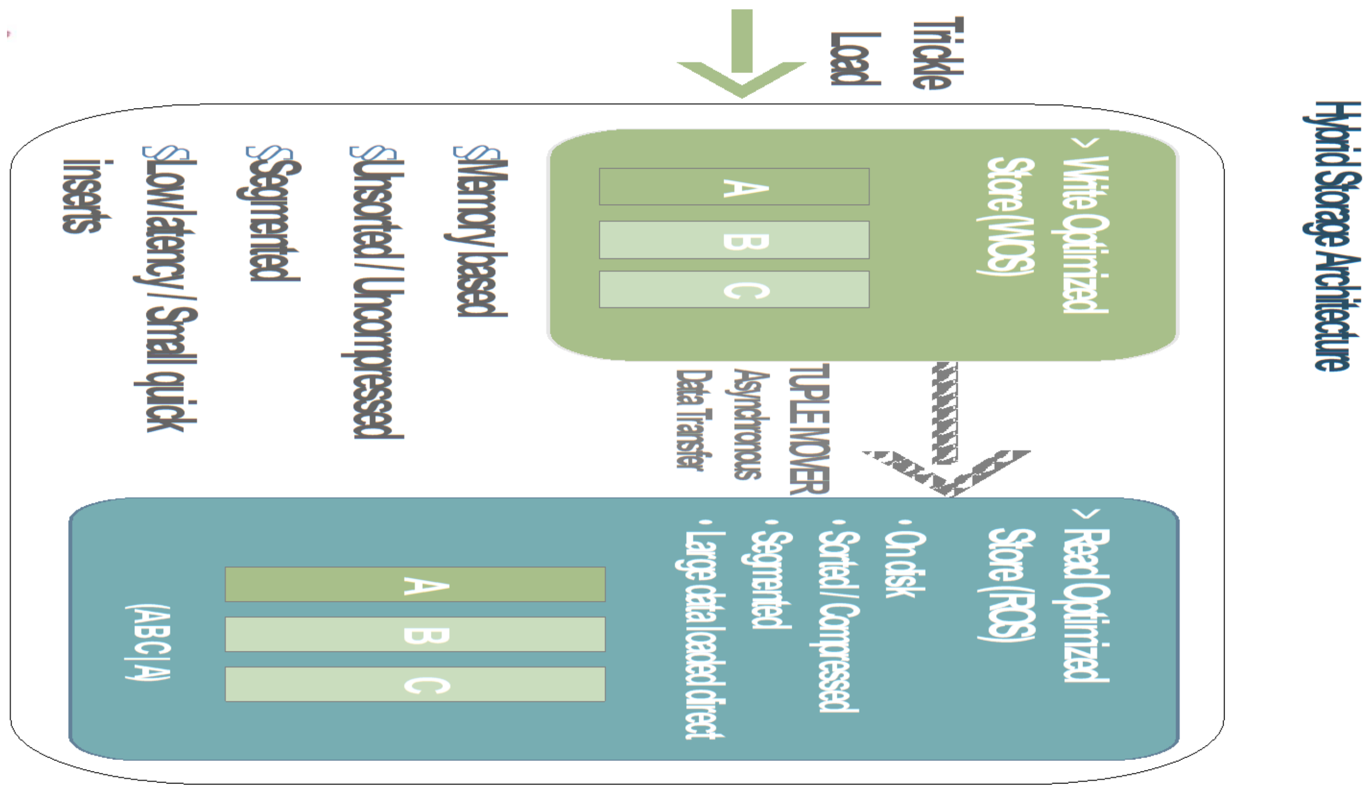


Figure 2: A join index from EMP3 to EMP1.

Column-stores: Updates?

- Typically a separate “write optimized store”



Vertica

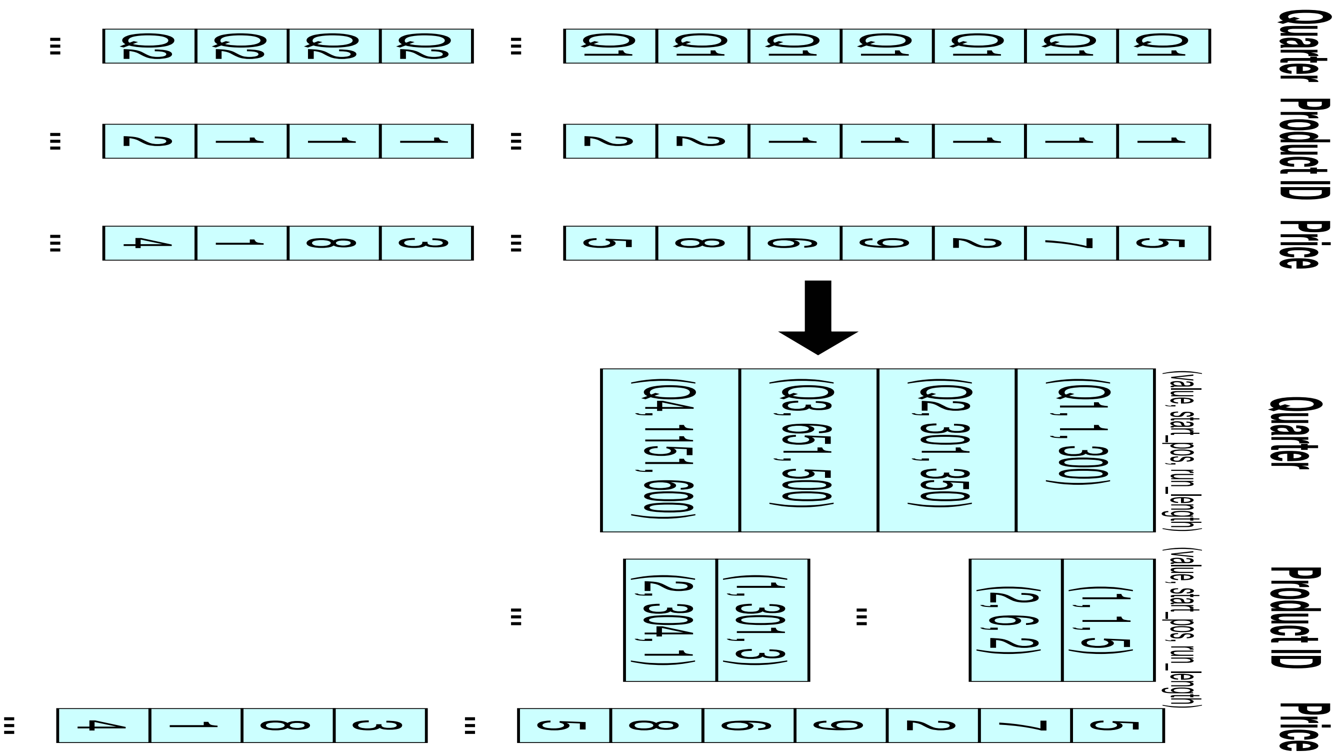
Material from Tutorial on Column-stores by Harizopoulos, Abadi, Boncz

Compression

- Compression can help reduce storage costs, and I/O Costs
  - But: decompression costs significant, and updates are more complicated
  - For row-stores, compression benefits lower because of heterogeneity

Compression in Column-stores

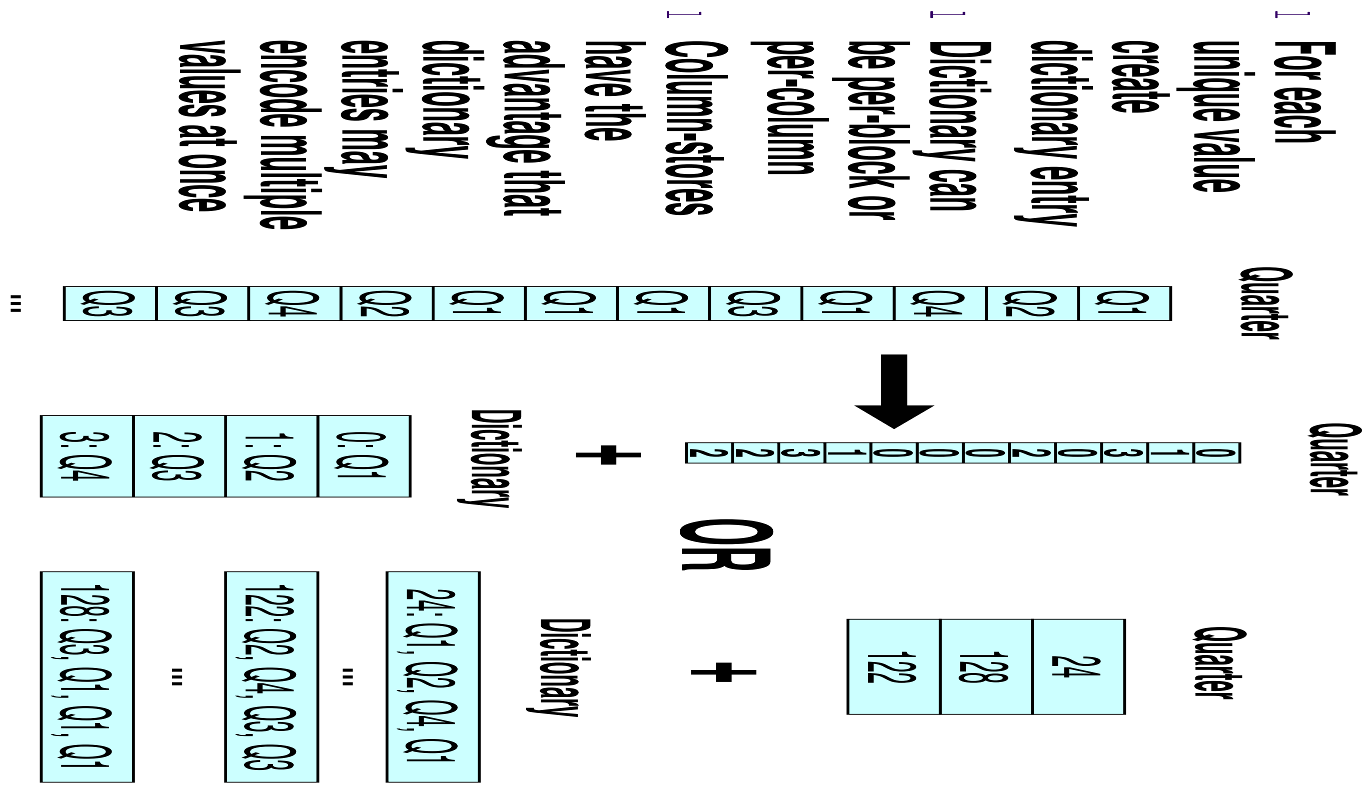
- Run-length Encoding
  - Works well for sorted data



Material from Tutorial on Column-stores by Harizopoulos, Abadi, Boncz

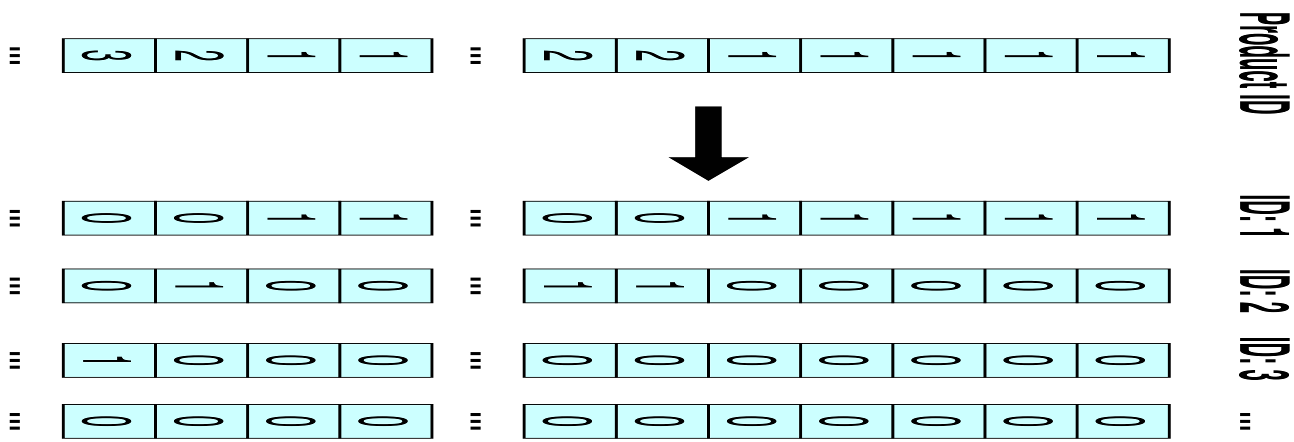
Compression in Column-stores

- Dictionary Encoding



Compression in Column-stores

- Bit-vector 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

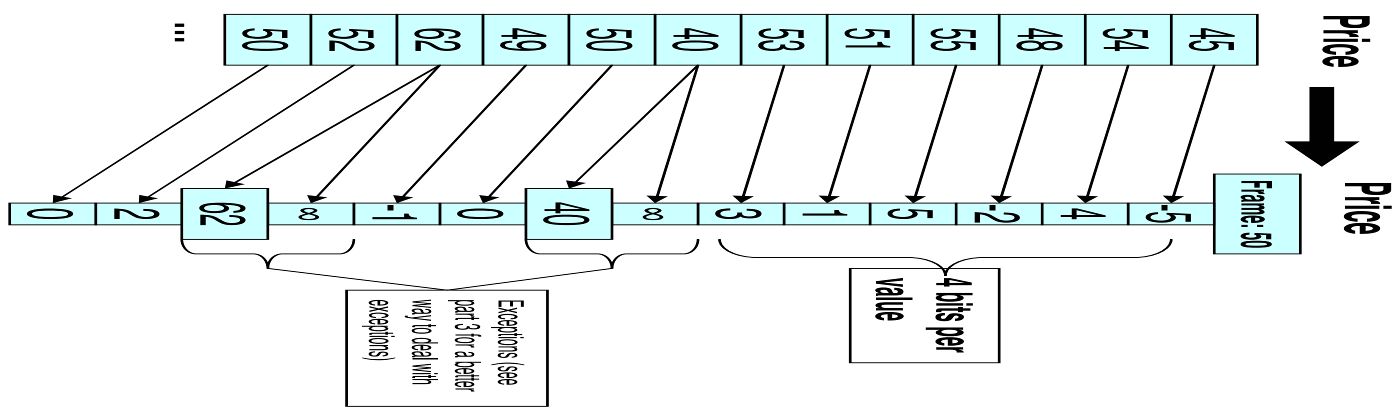


Compression in Column-stores

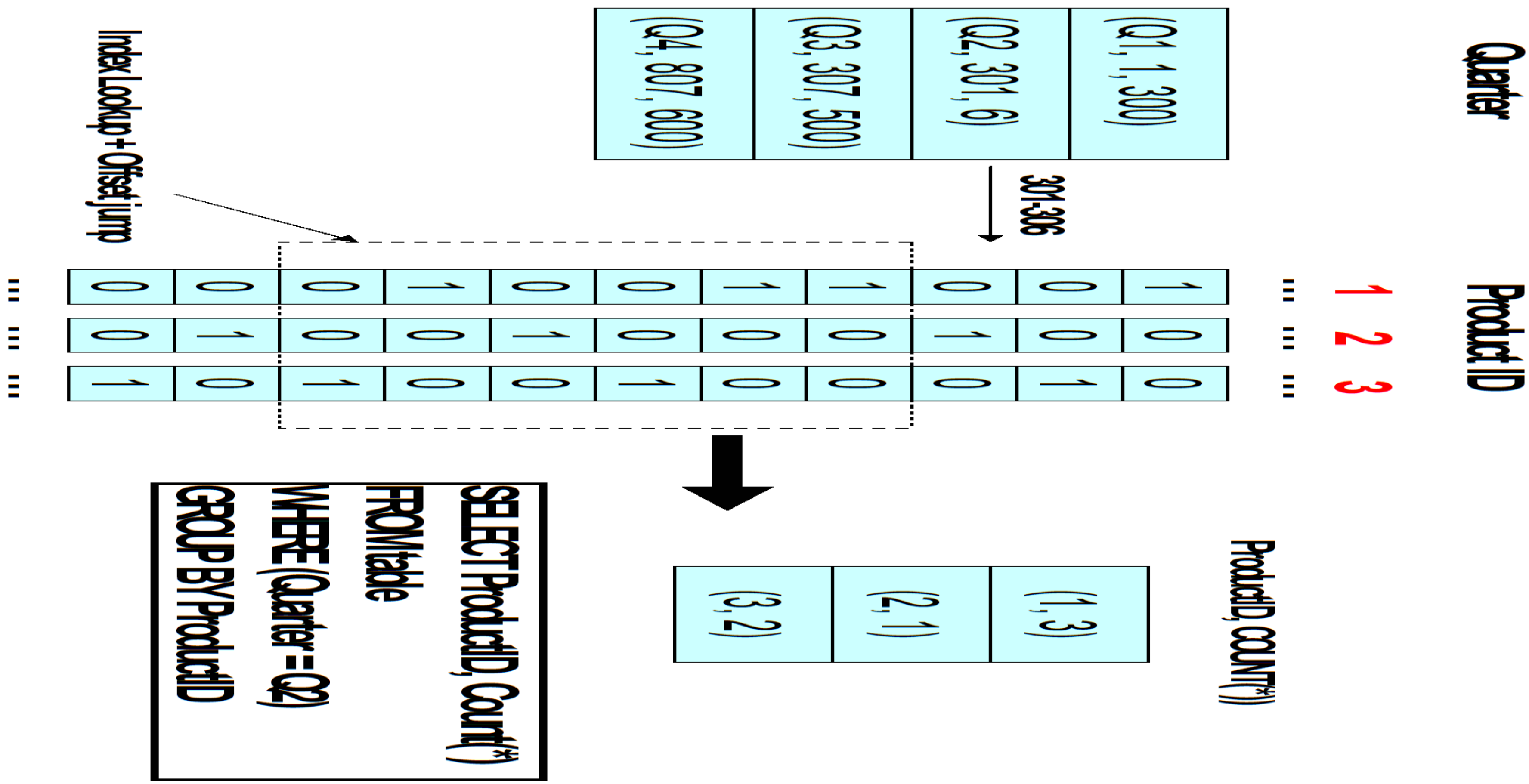
- Frame of Reference Encoding

  - Encodes values as b bit offset from chosen frame of reference
  - Special escape code (e.g. all bits set to 1) indicates a difference larger than can be stored in b bits
    - After escape code, original (uncompressed) value is written

"Compressing Relations and Indexes" Goldstein, Ramakrishnan, Shaft, ICDE'98



Execution on Compressed Data: Example



Execution on Compressed Data

- Clear benefits in specific cases, but too many combinations and special cases

$$\begin{array}{|c|}\hline 42\\\hline 36\\\hline 42\\\hline 44\\\hline 38\\\hline\end{array} \bowtie \begin{array}{|c|}\hline 38\\\hline 42\\\hline 46\\\hline\end{array} = \begin{array}{|c|c|}\hline 1 & 2\\\hline 3 & 2\\\hline 5 & 1\\\hline\end{array}$$

Join in C-Store may return IDs of tuples that match

```
NLJoin(Predicate q, Column c1, Column c2)
 if c1 is not compressed and c2 is not compressed
   for each value valc1 with position i in c1 do
     for each value valc2 with position j in c2 do
       if q(valc1,valc2) then output-left: (i), output-right: (j)
     end
   end
 if c1 is not compressed and c2 is RLE compressed
   for each value valc1 with position i in c1 do
     for each triple t with val v,startpos j and runlen k in c2
       if q(valc1,v) then:
         output-left: t,
         output-right: (j ... j+k-1)
     end
   end
 if c1 is not compressed and c2 is bit-vector compressed
   for each value valc1 with position i in c1 do
     for each value valc2 with bitstring b in c2 do
       //assume that there are num '1's in b
       if q(valc1,valc2) then output
         output-left: new RLE triple (NULL,i,num),
         output-right: b
     end
   end
 etc. etc. for every possible combination of encoding types
```

Execution on Compressed Data

- Compressed data divided into compressed "blocks"
  - e.g., for RLE, a block is single triple: (value, start_pos, run_length)
- Compressed block API allows extracting information from a block
  - getNext() or asArray() used for decompressing the data

Properties	Iterator Access	Block Information
isOneValue()	getNext()	getSize()
isValueSorted()	asArray()	getStartValue()
isPosContig()		getEndPosition()

Table 1: Compressed Block API

Execution on Compressed Data

- Use the APIs to write more generic code

```
COUNT(COLUMN c1)
 b = GET NEXT COMPRESSED BLOCK FROM c1
 WHILE b IS NOT NULL
   IF b.ISONEVALUE()
     x = FETCH CURRENT COUNT FOR b.GETSTARTVAL()
     x = x + b.GETSIZE()
   ELSE
     a = b.ASARRAY()
     FOR EACH ELEMENT i IN a
       x = FETCH CURRENT COUNT FOR i
       x = x + 1
   b = GET NEXT COMPRESSED BLOCK FROM c1
```

Figure 2: Pseudocode for Simple Count Aggregation

Encoding Type	**Sorted?**	**1 value?**	**Pos. contig.?**
RLE	yes	yes	yes
Bit-string	yes	yes	no
Null Supp.	no/yes	no	yes
Lempel-Ziv	no/yes	no	yes
Dictionary	no/yes	no	yes
Uncompressed	no/yes	no	no/yes

Outline

- Basics
- PAX: Within-page Columnar Storage
- Compression in Column-Stores
- Dremel: Storing Hierarchical Data
- Delta Lake: Storage Issues in Data Lakes

Motivation

- MapReduce good for analysis of large-scale data, but not appropriate for ad hoc (esp. aggregate) queries
- Much of the data is hierarchical (nested) and sparse, but with a schema (i.e., like JSON data)

- Dremel built at Google to address these use cases
  - A nested columnar storage format
  - In situ processing, i.e., process data in place
  - A distributed "serving tree" to propagate a query within the storage layer

Nested Columnar Storage

- Schema format from “protocol buffers”
  - Used for messages
  - Can handle ”lists” and “maps”

```
message Document {
  required int64 DocId;
  optional group Links {
    repeated int64 Backward;
    repeated int64 Forward; }
  repeated group Name {
    repeated group Language {
      required string Code;
      optional string Country; }
    optional string Url; }}
```

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”
	    },
	    …
	}

Nested Columnar Storage

- Schema format from “protocol buffers”
  - Used for messages
  - Can handle ”lists” and “maps”

```
message Document {
  required int64 DocId;
  optional group Links {
    repeated int64 Backward;
    repeated int64 Forward; }
  repeated group Name {
    repeated group Language {
      required string Code;
      optional string Country; }
    optional string Url; }}
```

r_1

```
DocId: 10
Links
  Forward: 20
  Forward: 40
  Forward: 60
Name
  Language
    Code: 'en-us'
    Country: 'us'
  Language
    Code: 'en'
  Url: 'http://A'
Name
  Url: 'http://B'
Name
  Language
    Code: 'en-gb'
    Country: 'gb'
```

r_2

```
DocId: 20
Links
  Backward: 10
  Backward: 30
  Forward:  80
Name
  Url: 'http://C'
```

Nested Columnar Storage

```
message Document {
  required int64 DocId;
  optional group Links {
    repeated int64 Backward;
    repeated int64 Forward; }
  repeated group Name {
    repeated group Language {
      required string Code;
      optional string Country; }
    optional string Url; }}
```

DocId
10
20

Links.Forward	
20	0
40	1
60	1
80	0

```
DocId: 10                r1
Links
  Forward: 20
  Forward: 40
  Forward: 60
Name
  Language
    Code: 'en-us'
    Country: 'us'
  Language
    Code: 'en'
  Url: 'http://A'
Name
  Url: 'http://B'
Name
  Language
    Code: 'en-gb'
    Country: 'gb'
```

```
DocId: 20                r2
Links
  Backward: 10
  Backward: 30
  Forward:  80
Name
  Url: 'http://C'
```

Links.Backward
10
30

N.L.Code
‘en-us’
‘en’
‘en-gb’

Nested Columnar Storage

```
message Document {
  required int64 DocId;
  optional group Links {
    repeated int64 Backward;
    repeated int64 Forward; }
  repeated group Name {
    repeated group Language {
      required string Code;
      optional string Country; }
    optional string Url; }}
```

```
DocId: 10                r1
Links
  Forward: 20
  Forward: 40
  Forward: 60
Name
  Language
    Code: 'en-us'
    Country: 'us'
  Language
    Code: 'en'
  Url: 'http://A'
Name
  Url: 'http://B'
Name
  Language
    Code: 'en-gb'
    Country: 'gb'
```

```
DocId: 20                r2
Links
  Backward: 10
  Backward: 30
  Forward:  80
Name
  Url: 'http://C'
```

DocId
10
20

Links.Forward	RL
20	0
40	1
60	1
80	0

Links.Backward	RL
NULL	0
10	0
30	1

N.L.Code	RL	DL
'en-us'	0	2
'en'	2	2
NULL	1	1
'en-gb'	1	2
NULL	0	1

Nested Columnar Storage

```
message Document {
  required int64 DocId;
  optional group Links {
    repeated int64 Backward;
    repeated int64 Forward; }
  repeated group Name {
    repeated group Language {
      required string Code;
      optional string Country; }
    optional string Url; }}
```

```
DocId: 10                r1
Links
  Forward: 20
  Forward: 40
  Forward: 60
Name
  Language
    Code: 'en-us'
    Country: 'us'
  Language
    Code: 'en'
  Url: 'http://A'
Name
  Url: 'http://B'
Name
  Language
    Code: 'en-gb'
    Country: 'gb'
```

```
DocId: 20                r2
Links
  Backward: 10
  Backward: 30
  Forward:  80
Name
  Url: 'http://C'
```

DocId

value	r	d
10	0	0
20	0	0

Name.Url

value	r	d
http://A	0	2
http://B	1	2
NULL	1	1
http://C	0	2

Links.Forward

value	r	d
20	0	2
40	1	2
60	1	2
80	0	2

Links.Backward

value	r	d
NULL	0	1
10	0	2
30	1	2

Name.Language.Code

value	r	d
en-us	0	2
en	2	2
NULL	1	1
en-gb	1	2
NULL	0	1

Name.Language.Country

value	r	d
us	0	3
~~NULL~~	~~2~~	~~2~~
NULL	1	1
gb	1	3
NULL	0	1

No need to store NULLs
$d < 3$ ➔ NULL

Why not just use a bitmap for NULLs?
→ The actual DL number needed for reconstruction

Reconstruction

DocId

value	r	d
10	0	0
20	0	0

Name.Url

value	r	d
http://A	0	2
http://B	1	2
NULL	1	1
http://C	0	2

Links.Forward

value	r	d
20	0	2
40	1	2
60	1	2
80	0	2

Links.Backward

value	r	d
NULL	0	1
10	0	2
30	1	2

Name.Language.Code

value	r	d
en-us	0	2
en	2	2
NULL	1	1
en-gb	1	2
NULL	0	1

Name.Language.Country

value	r	d
us	0	3
NULL	2	2
NULL	1	1
gb	1	3
NULL	0	1

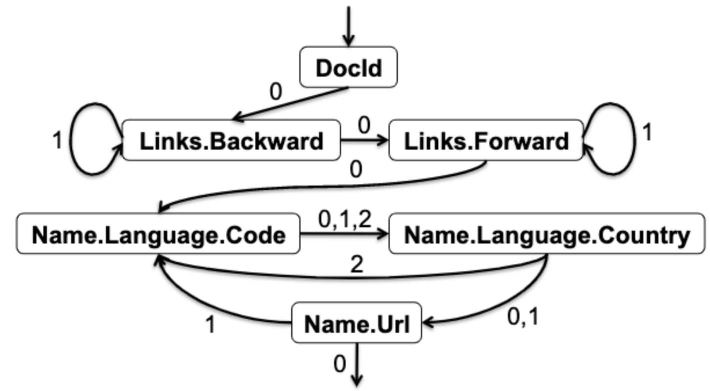


Figure 4: Complete record assembly automaton. Edges are labeled with repetition levels.

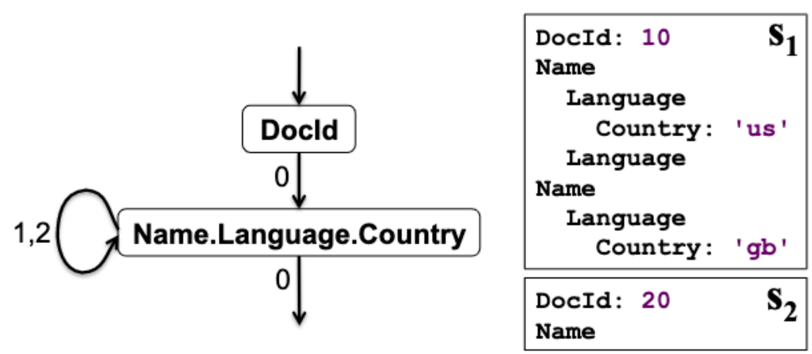


Figure 5: Automaton for assembling records from two fields, and the records it produces

Query Execution

- Only supports "one-pass aggregation" queries (i.e., no joins)
- An aggregation query split up across all "tablets", i.e., horizontal partitions of the table

- Experimental results showing queries can be executed in interactive times on disk-resident datasets up to a trillion records

Developments since 2010

- From the retrospective paper in 2020: Dremel: A Decade of Interactive SQL Analysis at Web Scale (”test-of-time award”)
- Several other formats proposed since then: Parquet (same as Dremel), ORC, Apache Arrow
  - ORC and Arrow keep track of number of repeated entries and explicit ”presence” bits

DocId

value	r	d
10	0	0
20	0	0

Name.Url

value	r	d
http://A	0	2
http://B	1	2
NULL	1	1
http://C	0	2

Links.Forward

value	r	d
20	0	2
40	1	2
60	1	2
80	0	2

Links.Backward

value	r	d
NULL	0	1
10	0	2
30	1	2

Name.Language.Code

value	r	d
en-us	0	2
en	2	2
NULL	1	1
en-gb	1	2
NULL	0	1

Name.Language.Country

value	r	d
us	0	3
NULL	2	2
NULL	1	1
gb	1	3
NULL	0	1

DocId

value	p
10	true
20	true

Name

len
3
1

Name.Url

value	p
http://A	true
http://B	true
	false
http://C	true

Name.Language

len
2
0
1
0

Name.Language.Code

value	p
en-us	true
en	true
en-gb	true

Name.Language.Country

value	p
us	true
	false
gb	true

Figure 7: Columnar representation of the data in Figure 5 showing length (len) and presence (p)

Developments since 2010

- From the retrospective paper in 2020: Dremel: A Decade of Interactive SQL Analysis at Web Scale ("test-of-time award")
- Several other formats proposed since then: Parquet (same as Dremel), ORC, Apache Arrow
  - ORC and Arrow keep track of number of repeated entries and explicit "presence" bits
- Google BigQuery now uses Capacitor
  - Very similar to Dremel, with some improvements in compression, etc.
- Authors note several open problems in this space, especially in understanding tradeoffs and dealing with heterogeneous data

Developments since 2010

- From the retrospective paper in 2020: Dremel: A Decade of Interactive SQL Analysis at Web Scale (”test-of-time award”)

- Other things of note:
  - Disaggregating storage/memory and compute beneficial in the long run
  - In situ data analysis important, but makes optimization hard
  - Need “shuffle” in order to improve query processing (e.g., to support joins)
  - Fully distributed query processing runs into issues – shifted to a more centralized approach

Outline

- Basics
- PAX: Within-page Columnar Storage
- Compression in Column-Stores
- Dremel: Storing Hierarchical Data
- Delta Lake: Storage Issues in Data Lakes

Motivation

- Cloud object stores increasingly used for data lake storage: Amazon S3, Azure Blob Storage, Google Cloud Storage, etc.
    - Data usually stored as Parquet or ORC columnar formats
- Hard to guarantee ACID properties – no multi-object atomic updates
- Different consistency models offered (e.g., ”read your writes” by S3)
- Very high latencies for interactive querying, API limitations (e.g., when running LIST against S3)
- Distributed file systems (like HDFS) also suffer from some of these issues

Existing Approaches to Table Storage

- Directories of files
  - Each table stored as a collection of objects (e.g., Parquet files)
  - No atomicity across multiple objects, eventual consistency, poor performance, no support for operations like table versioning, auditing, etc.
- Custom Storage Engines
  - Manage metadata in a layer above, that is strongly consistent
  - Basically treat the cloud storage as a disk
  - Challenges: All I/O operations need to go through metadata service, the metadata service layer can be hard to build efficiently
- Metadata in Object Stores (Delta Lake Approach)
  - Move metadata and transaction log into the object store itself
  - Challenges?

Delta Table Storage Format

- Each ”logical table” partitioned (if desired) and stored as Parquet files
- Logs generated as a JSON objects, periodically converted into Parquet format

Each log record object contains an array of actions:

- Change metadata
- Add or remove files
- Add provenance information
- Additional information for specific use cases

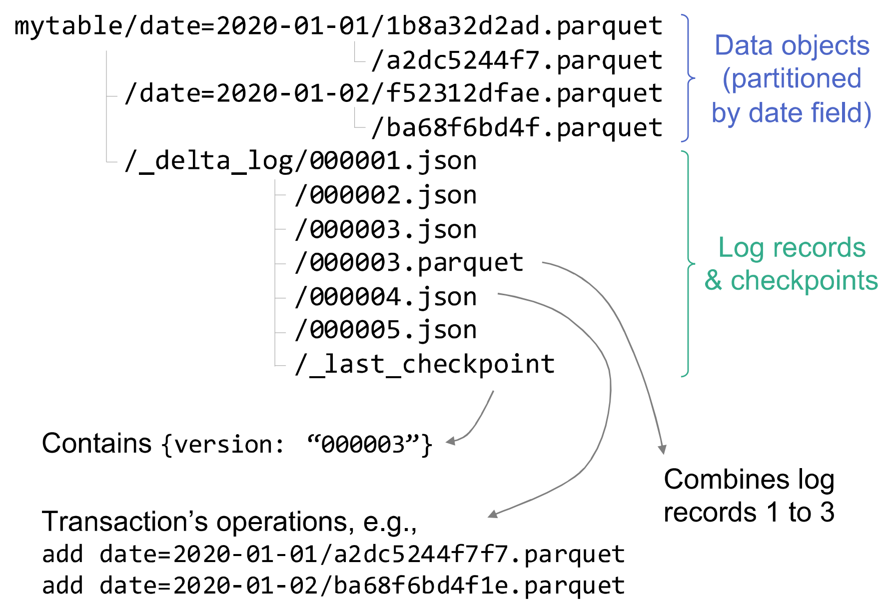


Figure 2: Objects stored in a sample Delta table.

Reading a Table

- Read the _last_checkpoint, and any other log files after that
- Figure out which data objects need to be read, using the metadata and statistics
- Read the relevant data objects, possible waiting (due to eventual consistency)

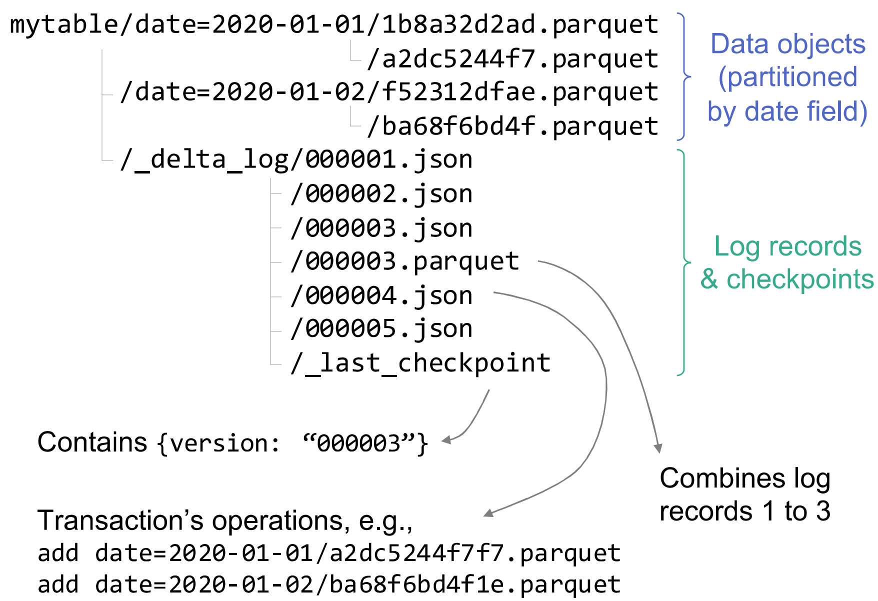


Figure 2: Objects stored in a sample Delta table.

Write Transactions

- Read up to the latest log record (say *r*) – we will try to write log record *r+1*
- Read data at table version *r*, and write new data objects into new files
- Attempt to write *r+1.json* – this needs to be atomic – if it fails, retry
- Optionally write a new checkpoint

Atomic write of *r+1.json:*

- *Google and Azure Cloud Storage support “put if absent”*
- *HDFS: can use “atomic rename” operation*
- *Amazon S3: Need a separate coordination service*

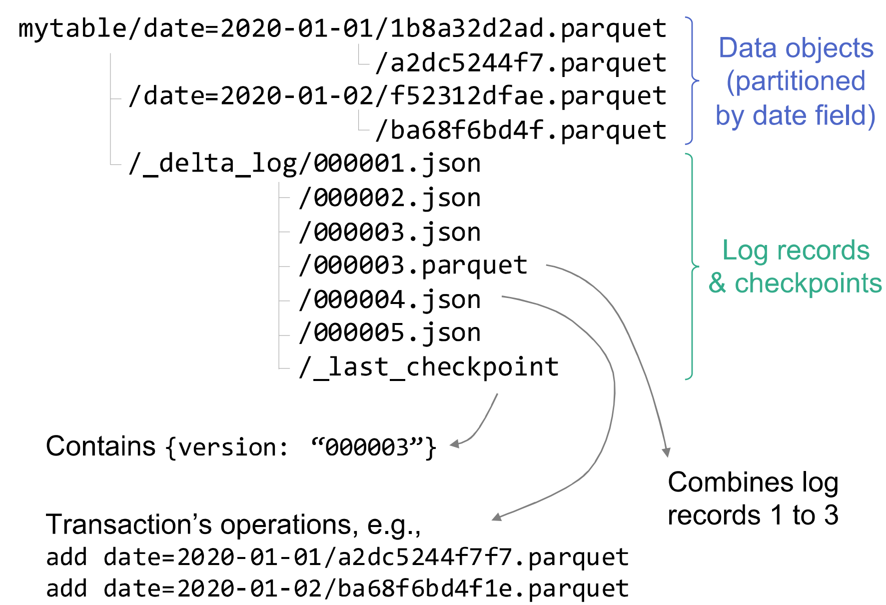


Figure 2: Objects stored in a sample Delta table.

More…

- Writes are serialized, but reads provide a snapshot (but not necessarily the latest version)
- No transactions across tables
- Transaction rates bottlenecked by the put-if-absent operations
- Time travel and rollbacks
  - Delta Lake data objects and log are "immutable", so easy to retrieve a past snapshot of the data
  - Can set retention periods on a per table basis
- Efficient UPSERT/DELETE/MERGE
  - Can use add/remove to efficiently support updates or deletes
- Streaming ingest and consumption
  - Can write small objects to start with, and then compact them in background
  - Could potentially avoid having to run a separate message bus altogether

More…

- Data Layout Optimization
  - Compact small objects in the background
  - Z-Ordering by multiple attributes (to make it easy to run select queries against multiple attributes)
  - Potentially build new indexes
- Audit history is naturally available
- Schema evolution and enforcement
  - Can update schemas in the background for older objects
- Connectors to other query and ETL engines
  - Special format of Delta Lakes requires specialized code
  - Can use “symlink manifest files” in some cases

Use Cases

- Simplify enterprise data architectures using a single system for many jobs rather than a separate system for each
    - So “one size does fit all”?
- Data engineering and ETL can be done directly against the Delta Lake
- Support for more efficient querying can handle some of the Warehousing use cases
- Compliance and reproducibility: through ability to delete old data easily, and time travel to retrieve past versions

Limitations

- Serializable transactions only within a single table
  - Technically a “delta lake table” could correspond to multiple “logical tables”
- Latencies for streaming operations
  - Still have to deal with cloud storage latencies
- Secondary indexes
  - Ongoing work on adding more types of indexes

Thoughts…

- Almost a throwback to the original motivation for a "shared data bank"
  - Hide all the complexity behind a logical abstraction that supports updates
  - Avoid many copies of the same dataset in different systems
- Likely to become increasingly common for data lakes
  - "Disaggregation" a common trend
  - Other data lakes support this kind of abstraction
  - Recent work on "self-organizing data containers" from MIT: looking into how to automatically reoptimize the data layouts

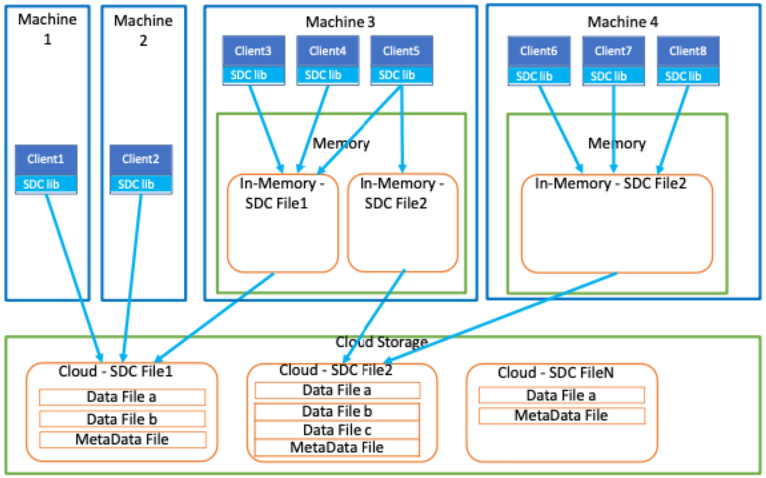


Figure 1: SDC Architecture: Client 1 and 2 access the same SDC file on cloud storage. Client 3 and 4 work on the same in-memory SDC file 2, which is backed by a cloud SDC file.

Open questions (and potential projects)

- Could this be used as the primary backend for an OLTP system? Why or why not?
- Impact of the partitioning granularity
  - Many small objects will make it easier to support updates, but penalize reads
  - Could automatically choose the partitioning granularity based on read/write pattern
- Materialized views to support efficient OLAP on top this
- Proper ”versioning” and “branching” in a setup like this
  - The only way to branch today is to make a copy of the entire table (CLONE)
- Fractured mirrors?
  - Two different systems may wish to simultaneously have the same table in different formats/layouts
  - Could that be pushed inside the abstraction? how would consistency work?
- Could use this abstraction to separate sensitive data from non-sensitive data automatically (for privacy and security)