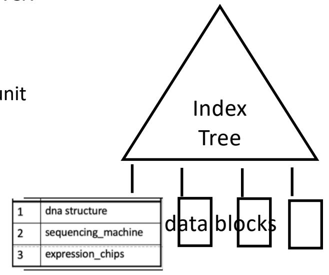
Storage Models

Objectives:

- Detail three storage models,
  - Row storage (conventional)
  - Column Storage
  - Parallel/Distributed Key-Value stores (NoSQL)
- And a first look at parallel/distributed database structure

Row Storage classic, conventional

- That's what we've been doing
  - Row is both a logical and physical unit
  - Primary key,
    - Basis of sorting rows on data blocks
    - Search key for the primary index

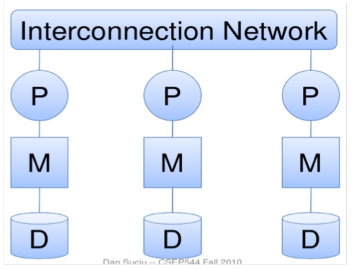


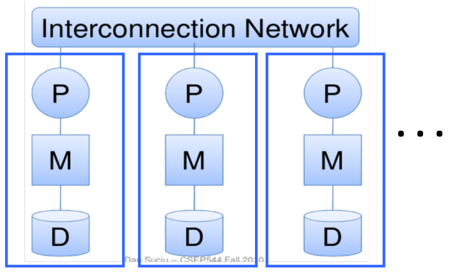
Horizontal Partition of Row Storage

- First introduction to parallel databases

Shared Nothing [DB] Architecture

aka *A cluster*





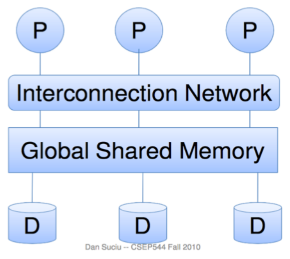
- Low cost, *commodity,* servers
  - Connected by a network
- Won (at least mind share)

Shared Nothing – Advantages

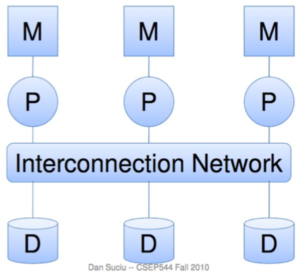
- Economical: uses commodity hardware
  - Rack mounted servers
- Most scalable
  - Minimizes interference by minimizing resource sharing
  - Memory and I/O bandwidth and capacity grow with the number of compute nodes.

Maybe later in the semester

- Shared Memory,



- Shared Disk



- My favorite: “Weird Machines”

Other Architectures…
people think they lost

but they actually live,
(they have a low profile)
they could come back

New Idea 1: Partition Data

- Partition Data: to split the storage of a table across the servers.
  - Horizontal Paritioning
  - Vertical Partitioning

3 Methods of Horizontal Partitioning

- Round Robin
- Hash
- Range

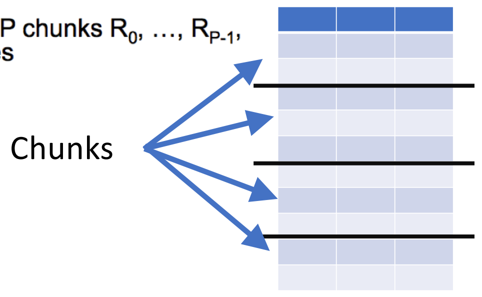
Horizontal Data Partitioning

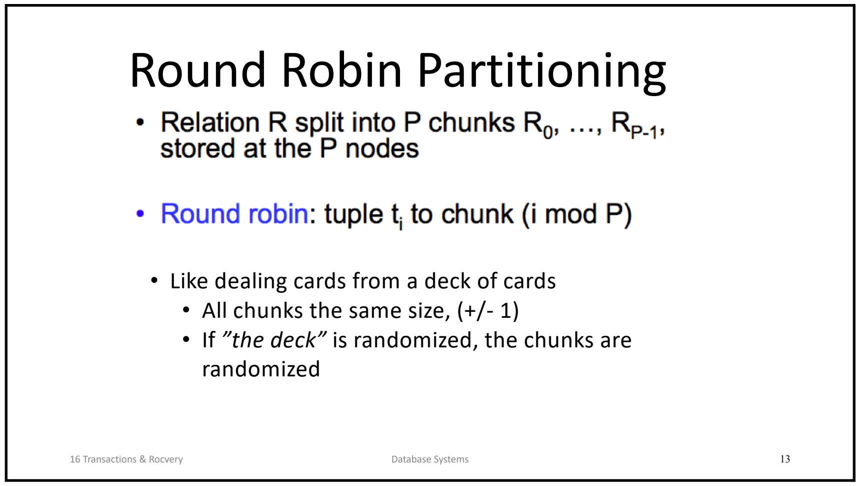
- Relation R split into P chunks $R_0, \ldots, R_{P-1}$, stored at the P nodes

Let t_j be a tuple in chunk R_i
Let a_j be the value or set of values
for an attribute or set of attribute for all t_j

Horizontal Data Partitioning

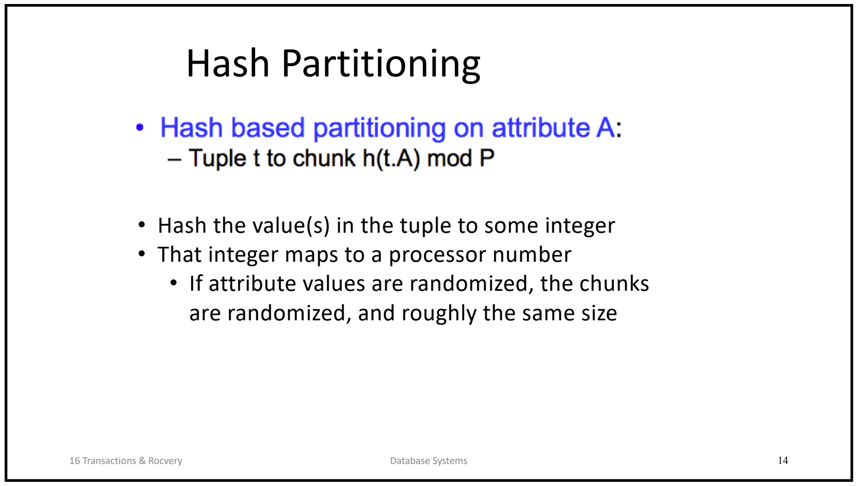
- Relation R split into P chunks $R_0, \ldots, R_{P-1}$, stored at the P nodes





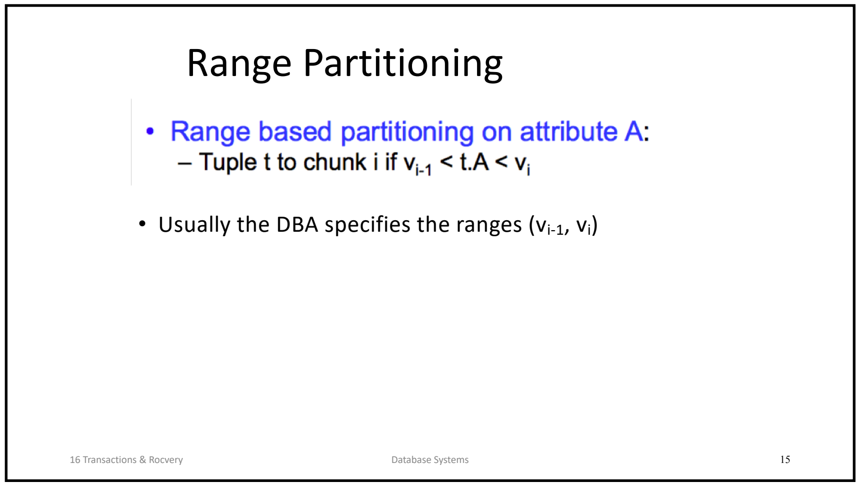
Round Robin Partitioning

- Relation R split into P chunks $R_0, \ldots, R_{P-1}$, stored at the P nodes
- Round robin: tuple t_i to chunk (i mod P)
  - Like dealing cards from a deck of cards
    - All chunks the same size, (+/- 1)
    - If *"the deck"* is randomized, the chunks are randomized



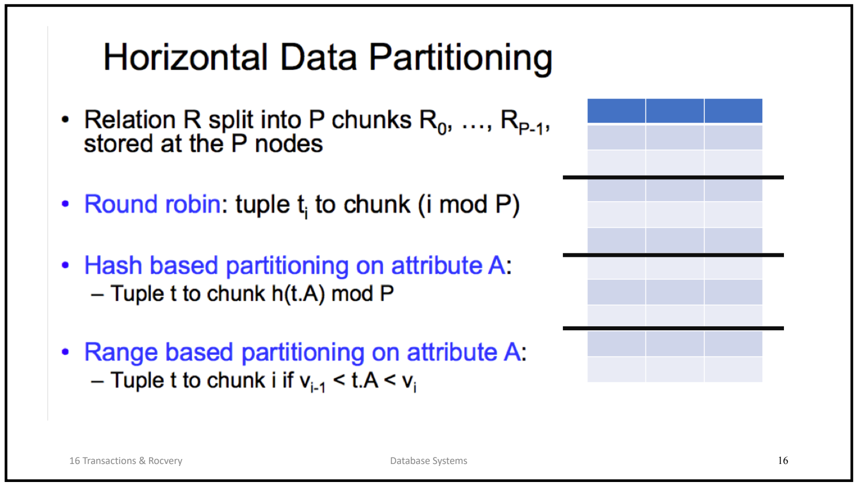
Hash Partitioning

- Hash based partitioning on attribute A:
  – Tuple t to chunk h(t.A) mod P
- Hash the value(s) in the tuple to some integer
- That integer maps to a processor number
  - If attribute values are randomized, the chunks are randomized, and roughly the same size



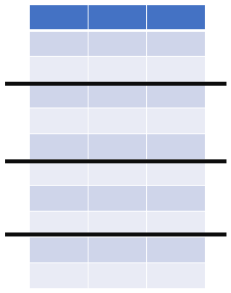
Range Partitioning

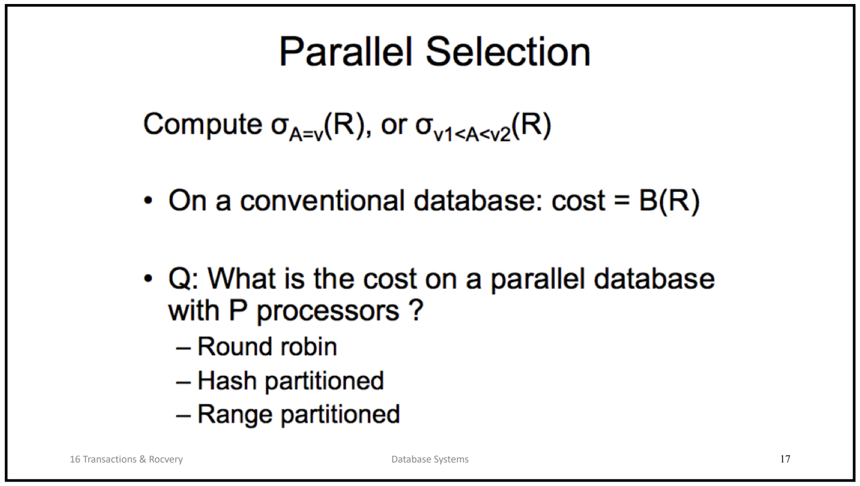
- Range based partitioning on attribute A:
  – Tuple t to chunk i if $v_{i-1} < t.A < v_i$
- Usually the DBA specifies the ranges (v_{i-1}, v_i)



Horizontal Data Partitioning

- Relation R split into P chunks $R_0, \ldots, R_{P-1}$, stored at the P nodes
- Round robin: tuple t_i to chunk (i mod P)
- Hash based partitioning on attribute A:
  – Tuple t to chunk h(t.A) mod P
- Range based partitioning on attribute A:
  – Tuple t to chunk i if $v_{i-1} < t.A < v_i$



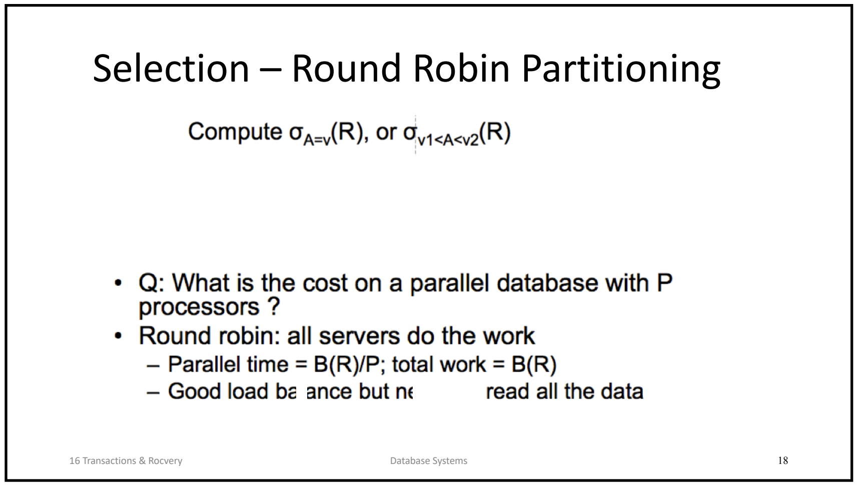


Parallel Selection

Compute $\sigma_{A=v}(R)$, or $\sigma_{v1<A<v2}(R)$

- On a conventional database: cost = B(R)
- Q: What is the cost on a parallel database with P processors ?
  - Round robin
  - Hash partitioned
  - Range partitioned

16 Transactions & Rocvery Database Systems 17

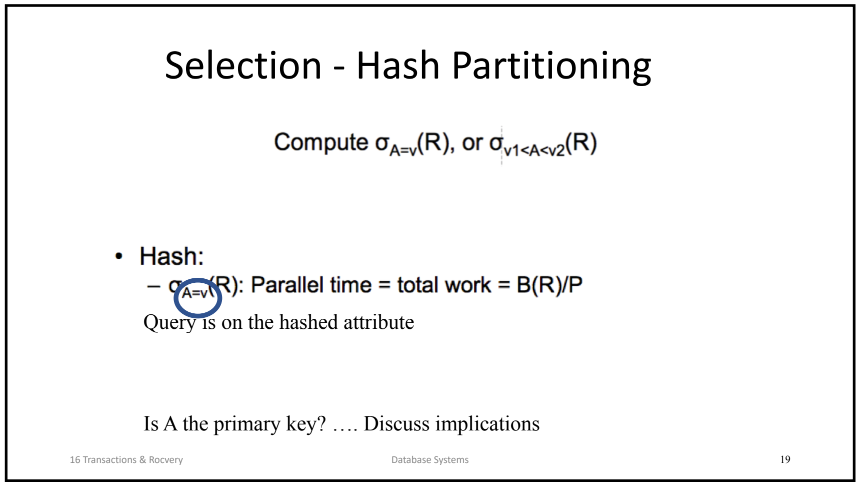


Selection – Round Robin Partitioning

Compute $\sigma_{A=v}(R)$, or $\sigma_{v1<A<v2}(R)$

- Q: What is the cost on a parallel database with P processors ?
- Round robin: all servers do the work
  - Parallel time = B(R)/P; total work = B(R)
  - Good load ba ance but ne read all the data

16 Transactions & Rocvery Database Systems 18



Selection - Hash Partitioning

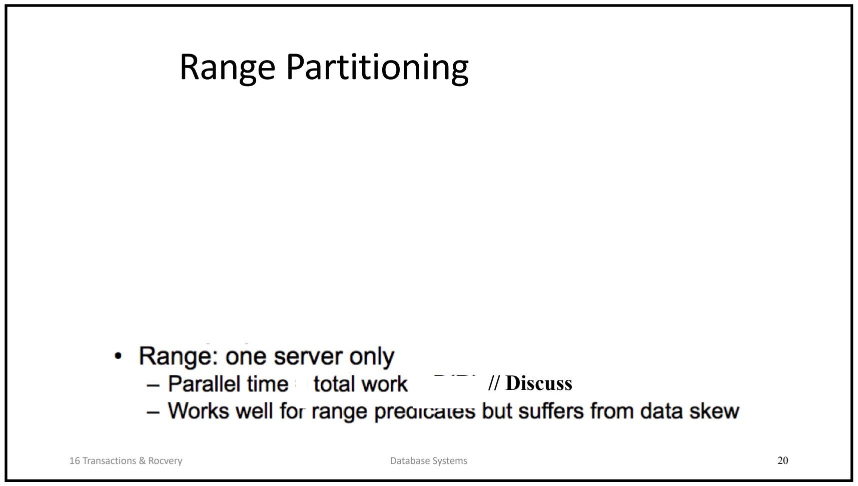
Compute $\sigma_{A=v}(R)$, or $\sigma_{v1<A<v2}(R)$

- Hash:
  - $\sigma_{A=v}(R)$: Parallel time = total work = B(R)/P

Query is on the hashed attribute

Is A the primary key? …. Discuss implications

16 Transactions & Rocvery Database Systems 19



Range Partitioning

- Range: one server only
  - Parallel time total work // **Discuss**
  - Works well for range predicates but suffers from data skew

16 Transactions & Rocvery Database Systems 20

Parallel Selection

- Q: What is the cost on a parallel database with P processors ?
- Round robin: all servers do the work
  - Parallel time = B(R)/P; total work = B(R)
  - Good load balance but needs to read all the data
- Hash:
  - $\sigma_{A=v}(R)$: Parallel time = total work = B(R)/P
  - $\sigma_{A\in[v1,v2]}(R)$: Parallel time = B(R)/P; total work = B(R)
- Range: one server only
  - Parallel time total work **// Discuss**
  - Works well for range predicates but suffers from data skew

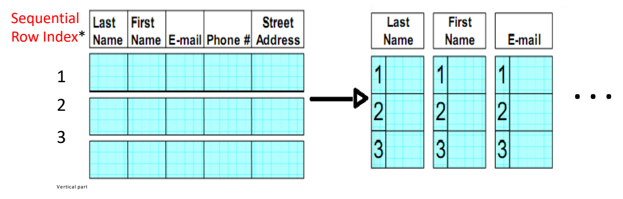
The Column Store Storage Model

Column Stores: The SQL entry to NoSQL Big Data

1. Column stores ***are not*** NoSQL, but you will see that in marketing
   - They *do* support Big Data
   - Marketing people don't get modas ponens

NoSQL → Big data
Column Store → Big data
So, marketing says -
Column Store → NoSQL **Not,** and *they are being called out on it.*

Vertical Partitioning → Column Store

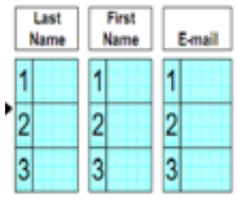


Horizontal Vertical

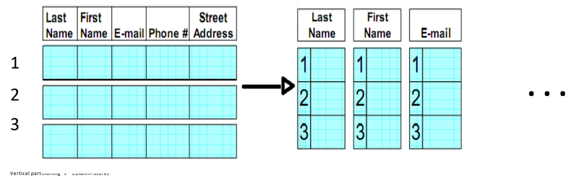
- *Row index (think in terms of an array) is not necessarily the primary key.
- No assurance a declared primary key embody the essential sequential property need to create a bitmap.

The Good

- Storage Space…
  - Use Bitmap index (only) as data storage
  - Use **Compressed** Bitmap index (only) as data storage



The Bad: Consider Insert(new_row)



How many disk accesses (writes)?
- Horizontal: [as few as] 1
- Vertical: at least the number of columns

… and then there is the transaction log

So:

- Transactional workload or analytic workload?

Star Schema: (a.k.a. OLAP schema)
SSBM benchmark derived from TPC-H
Current: "dimensional data modeling"

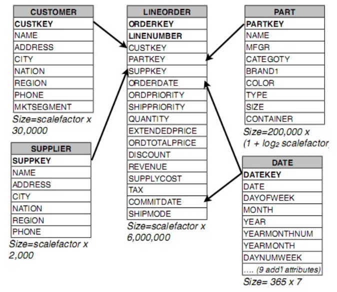


Figure 1: Schema of the SSBM Benchmark

Figure taken form [1]

28

The Very Good

```
SELECT  Sum(S.sales)
FROM sales_table S, time T, location L
WHERE L.city = 'Austin' AND
          T.year = 2019    AND
          S.locationkey = L.pk_loc AND
          S.timekey = T.pk_time
```

- How many columns must be read to process query? 7
  - L.city, L.pk_loc
  - T.year, T.pk_time
  - S.locationkey, S.timekey, S.sales
- In contemporary systems these are stored as compressed bitmaps.
- Compare to amount of data to read in a conventional horizontal store (?)

What actually get's read?

```
SELECT  Sum(S.sales)
FROM sales_table S, time T, location L
WHERE L.city = 'Austin' AND
          T.year = 2019    AND
          S.locationkey = L.pk_loc AND
          S.timekey = T.pk_time
```

- L.city, just the bitmap for 'Austin'
- T.year, just the bitmap for 2019

But to do the explicit joins we'll need all values (bitmaps) for

- S.locationkey, L.pk_loc
- S.timekey, T.pk_time

Similarly to compute the output Sum(S.sales), we will need the bitmaps for all values

Improved I/O not without cost:

```
SELECT  Sum(S.sales)
FROM sales_table S, time T, location L
WHERE L.city = 'Austin' AND
          T.year = 2019    AND
          S.locationkey = L.pk_loc AND
          S.timekey = T.pk_time
```

What needs to be joined?

- Seemingly unavoidable
  - S.locationkey = L.pk_loc
  - S.timekey = T.pk_time

But to actually compute the result

- For S, the sequential index of S.locationkey, S.timekey, S.sales,
  - must all be the same.
  - Each may take on many values

  → {many} x {many} x {many)
    - In bitmap can be done linear time AND
    - If Not in bitmap, but sorted, linear time merge
- Similarly,
  - T.year, T.pk_time
  - L.city, L.pk_loc
  - But just 1, value, Austin and 2019 have to match index with the other argument (from the same table)
  - → AND the bitmaps
- Hence,
  - 2 obvious joins in the SQL query
  - + 4 operations to assemble the output

Consider

```
SELECT  *
FROM sales_table S, time T, location L
WHERE L.city = 'Austin' AND
          T.year = 2019    AND
          S.locationkey = l.pk_loc AND
          S.timekey = T.pk_time
```

** all columns of the three tables

- Reading all columns, all values
- Reassemble columns into rows

Thus, Column Store vs. Row Store Tradeoff

For a given query

- Column store may require many fewer disk block reads than a horizontal store.
- Column store requires computation not needed by a horizontal store to assemble output.
- Note: More column reads → more work to assemble output.

Column Stores

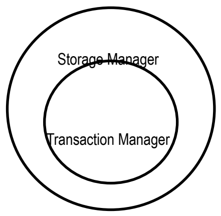
- From research
  - MonetDB (open source) [2002]
  - H-store and C-store ~[2005]
- To commercial practice
  - Sybase IQ [1995], bought by SAP… evolved to:
  - SAP HANA [2008], main-memory, cluster
  - HP Vertica [Vertica founded 2005 from C-store fork]
  - Sisense // a Tableau competitor, offers similar function but on multiple terabytes on a desktop
- "Proof" column store ≣ SQL RDBMS
  - MariaDB (mySQL fork) *Column Store version*
  - SQL Server (starting, 2016 V13), *columnstore indexes*

NoSQL Key Value Stores

1. Parallel/Distributed Storage
2. Notion of numbered Logical and Physical processors

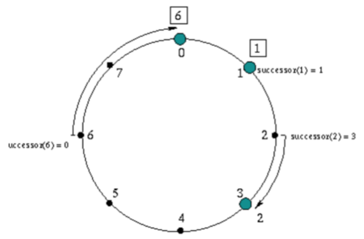
   (which will be ignored until near the last slide)
3. Each processor is replicated many times
   - Replicates contain the same data

NoSQL



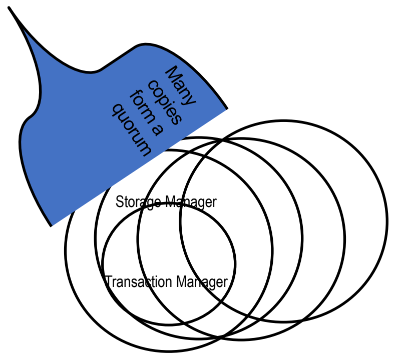
Chord Protocol [http://www.inf.ed.ac.uk/teaching/courses/ip/chord-desc.html]

The Chord protocol: given a key,
determine the node responsible for storing the key's value



Chord assigns hash keys to nodes in a way that doesn't need to change much as nodes join and leave the system. SIGCOMM '01 paper by Stoica et al.

NoSQL



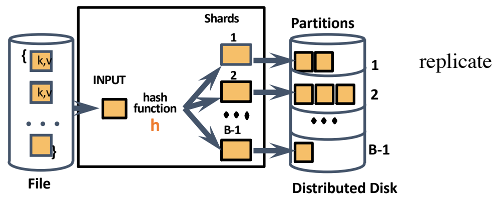
Key, Value model

Setname: {(key1, value1), (key2, value2) (key3, value3)…}

- Key_i Typically any string, but also OID (object id)
- store(key_i, $value_i$) will store (key_i, $value_i$) replicates in processor Chord(key_i)
  - the key is explicitly stored as its often also data
  - Data type of $value_i$ depends on the system, but can be anything
    - Json document
    - A nested set of (key, value) pairs

Hashing Shards, Replicate for Durability

- Hashing



Distributed File System

- Hadoop Distributed File System (HDFS)
  - pages/blocks ~~16~~ or 64 or 128 Mbytes
  - pages are compressed for storage
  - pages are replicated for fault tolerance
  - quorum consistency is the basis of transactions

NOSQL: Not as Different as "they" Would Like You to Believe

- E.g.
  - name{ (k1, dan), (k2, bob), (k3, bruce)}
  - salary{ (k1, $10), (k2, $1), (k3, $1000)}
  - title {(k1, professor), (k2, fry cook), (k3, chairman)}
- Employee

Id	Name	Salary	Title
1	dan	10	professor
2	bob	1	fry cook
3	bruce	1000	chairman
…			

Look familiar?

- E.g.
  - name{ (k1, dan), (k2, bob), (k3, bruce)}
  - salary{ (k1, $10), (k2, $1), (k3, $1000)}
  - title {(k1, professor), (k2, fry cook), (k3, chairman)}
- Employee

Id	Name	Salary	Title
1	dan	10	professor
2	bob	1	fry cook
3	bruce	1000	chairman
…			

How about now?

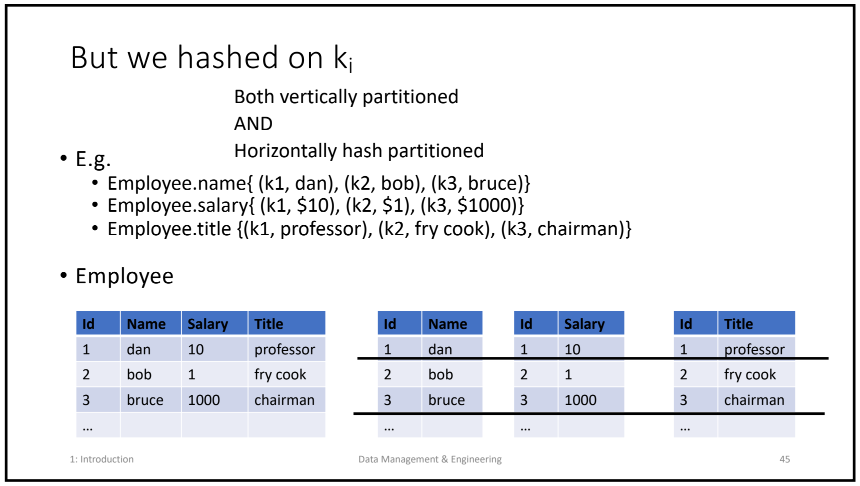
- E.g.
  - Employee.name{ (k1, dan), (k2, bob), (k3, bruce)}
  - Employee.salary{ (k1, $10), (k2, $1), (k3, $1000)}
  - Employee.title {(k1, professor), (k2, fry cook), (k3, chairman)}
- Employee

Id	Name	Salary	Title
1	dan	10	professor
2	bob	1	fry cook
3	bruce	1000	chairman
…			

Id	Name
1	dan
2	bob
3	bruce
…	

Id	Salary
1	10
2	1
3	1000
…	

Id	Title
1	professor
2	fry cook
3	chairman
…	



But we hashed on k_i

Both vertically partitioned
AND
Horizontally hash partitioned

- E.g.
  - Employee.name{ (k1, dan), (k2, bob), (k3, bruce)}
  - Employee.salary{ (k1, $10), (k2, $1), (k3, $1000)}
  - Employee.title {(k1, professor), (k2, fry cook), (k3, chairman)}
- Employee

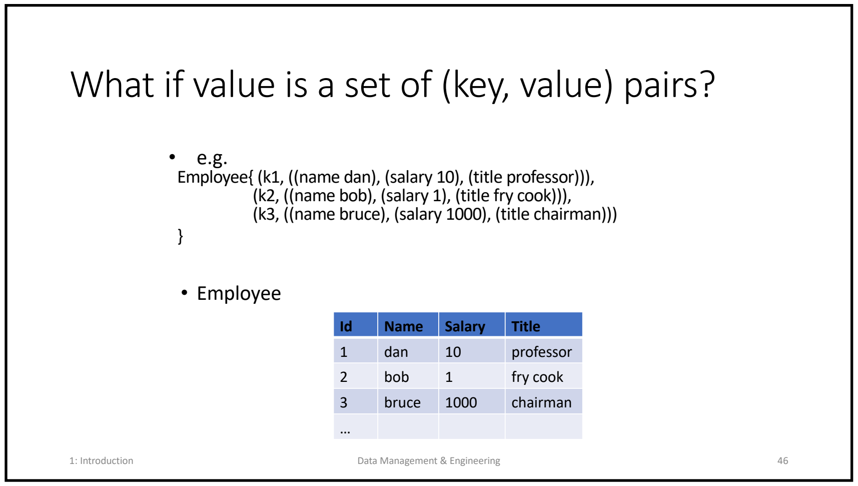
Id	Name	Salary	Title
1	dan	10	professor
2	bob	1	fry cook
3	bruce	1000	chairman
…			

Id	Name
1	dan
2	bob
3	bruce
…	

Id	Salary
1	10
2	1
3	1000
…	

Id	Title
1	professor
2	fry cook
3	chairman
…	

1: Introduction Data Management & Engineering 45



What if value is a set of (key, value) pairs?

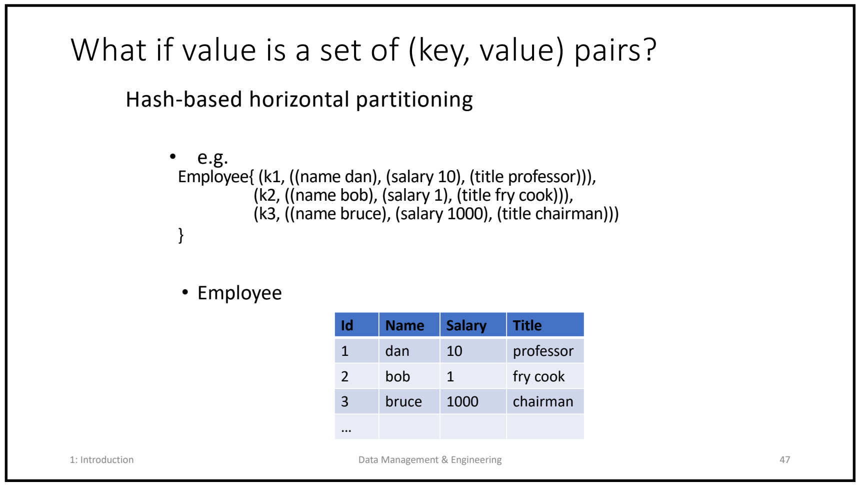
- e.g.

Employee{ (k1, ((name dan), (salary 10), (title professor))),
(k2, ((name bob), (salary 1), (title fry cook))),
(k3, ((name bruce), (salary 1000), (title chairman)))
}

- Employee

Id	Name	Salary	Title
1	dan	10	professor
2	bob	1	fry cook
3	bruce	1000	chairman
…			

1: Introduction Data Management & Engineering 46



What if value is a set of (key, value) pairs?

Hash-based horizontal partitioning

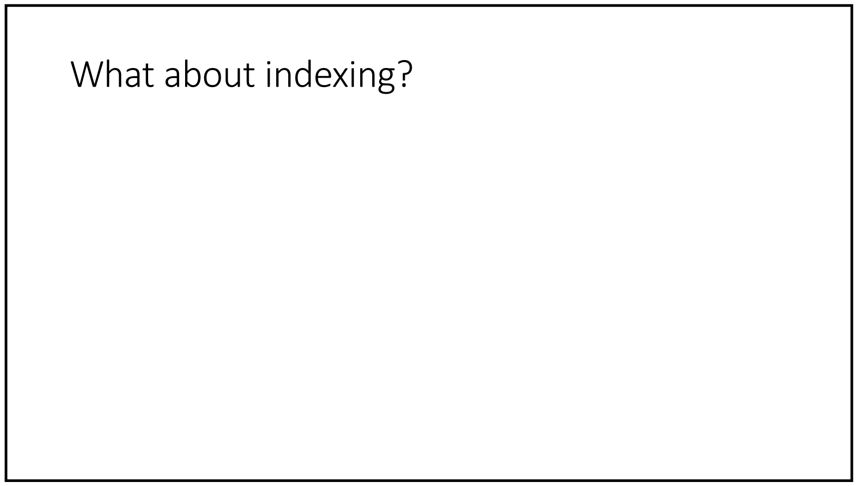
- e.g.

Employee{ (k1, ((name dan), (salary 10), (title professor))),
(k2, ((name bob), (salary 1), (title fry cook))),
(k3, ((name bruce), (salary 1000), (title chairman)))
}

- Employee

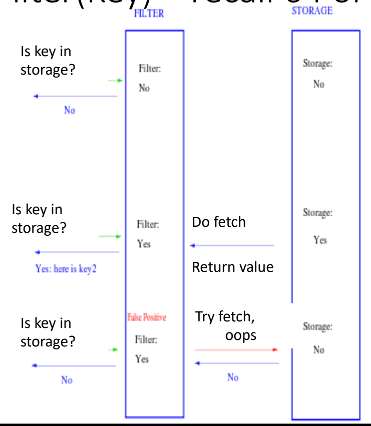
Id	Name	Salary	Title
1	dan	10	professor
2	bob	1	fry cook
3	bruce	1000	chairman
…			

1: Introduction Data Management & Engineering 47



What about indexing?

Bloom_Filter(Key) – recall 64 or 128mbyte pages



https://en.wikipedia.org/wiki/File:Bloom_filter_speed.svg

What about secondary indexes?

- Add more Bloom Filters
  - Declare the field used as a key

** pseudo code

CREATE Bloomfilter INDEX foo ON Employee.name(value)

- E.g.
  - Employee.name{ (k1, dan), (k2, bob), (k3, bruce)}
  - Employee.salary{ (k1, $10), (k2, $1), (k3, $1000)}
  - Employee.title {(k1, professor), (k2, fry cook), (k3, chairman)}
- Member(foo, “dan”)
  - Executes concurrently on all processors.

Id	Name
1	dan
2	bob
3	bruce
…	

Id	Salary
1	10
2	1
3	1000
…	

Id	Title
1	professor
2	fry cook
3	chairman
…	

** pseudo code

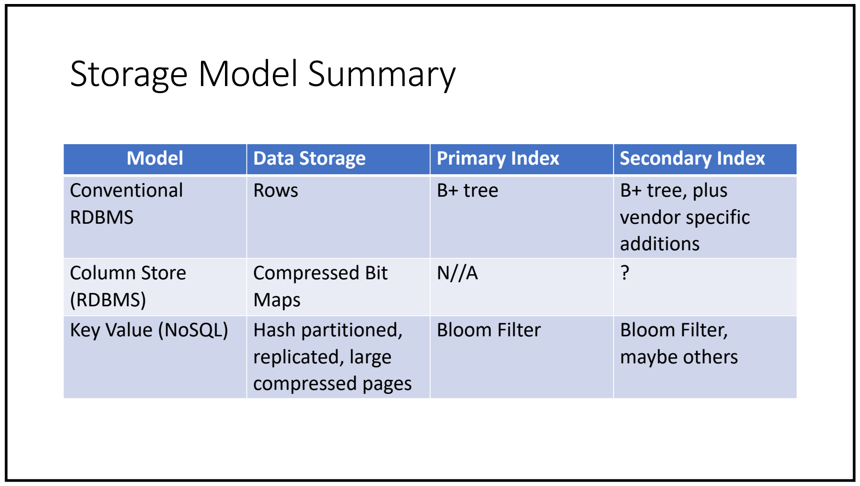
CREATE Bloomfilter INDEX foo ON Employee(WHERE subkey = “name”)

- e.g.

```
Employee{ (k1, ((name dan), (salary 10), (title professor))),
          (k2, ((name bob), (salary 1), (title fry cook))),
          (k3, ((name bruce), (salary 1000), (title chairman)))
}
```

- Employee

Id	Name	Salary	Title
1	dan	10	professor
2	bob	1	fry cook
3	bruce	1000	chairman
…			



Storage Model Summary

Model	Data Storage	Primary Index	Secondary Index
Conventional RDBMS	Rows	B+ tree	B+ tree, plus vendor specific additions
Column Store (RDBMS)	Compressed Bit Maps	N//A	?
Key Value (NoSQL)	Hash partitioned, replicated, large compressed pages	Bloom Filter	Bloom Filter, maybe others