MSiA-413 Introduction to Databases and Information Retrieval

Lecture 17
Partitioning, NoSQL databases & Big Data

Instructor: Nikos Hardavellas

Slides adapted from Steve Tarzia, A. Ailamaki, R. Elmasri, S. Navathe, EnterpriseDB, NTT, G. Trajcevski, R. Ramakrishnan, J. Gehrke

A single computer can only fit so much data

Larger, but slower

delay		*capacity*
0.3ns	CPU Registers	1 kB (kilobyte)
5ns	CPU Caches (L2)	16 MB
50ns	Random Access Memory (RAM)	16 GB
100µs	Flash Storage (SSD)	1 TB
5ms	Magnetic Disk	8 TB

What happens if we need to store hundreds of terabytes or thousands of terabytes (petabytes)?

A database server @ NU

- 264 fast (10k RPM) magnetic disks (for production)
- 56 slow (7200 RPM) magnetic disks (for backup)
- ~150 TB storage capacity
- Comprised of 6 physical chassis (boxes) in one big cabinet, about the size of a coat closet

This is about as big as you can grow one machine today (about 100TB)

Front view

SAS cabling in back

Truly "Big Data" requires many machines

Why not just add more disks to one machine?

- Each CPU has limited *input/output* capacity
  - Because there are only a few dozen I/O "pins" (wires) on the CPU
- Each CPU has limited *memory* for fast access to data
- Each CPU has limited *processing* (arithmetic) capacity

One way to use multiple machines in a database is through table partitioning

Why to partition tables?

- Query/DML Performance
  - Distribute queries across partitions and machines
  - Use of fast memory; a partition may fit in memory, a full table may not
  - Optimizations: partition elimination (pruning), partition-wise joins, aggregates
- Manageability
  - Smaller data is easier to manage
  - Partition-wise utility commands
  - Easy bulk load and delete
- Data storage based on partition properties
  - "hot" and "cold" partitions
  - Foreign partitions

Vertical vs. horizontal data partitioning

- Vertical partitioning: each partition has some columns
  - More common in distributed database systems
- Horizontal partitioning: each partition has some rows
  - More common in parallel relational database systems

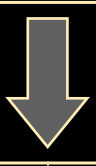
Vertical Partitioning

Horizontal Partitioning

Vertical partitioning

Resumes

SSN	Name	Address	Resume	Picture
234234	Mary	Huston	Clob1…	Blob1…
345345	Sue	Seattle	Clob2…	Blob2…
345343	Joan	Seattle	Clob3…	Blob3…
234234	Ann	Portland	Clob4…	Blob4…



T1

SSN	Name	Address
234234	Mary	Huston
345345	Sue	Seattle
. . .		

T2

SSN	Resume
234234	Clob1…
345345	Clob2…

T3

SSN	Picture
234234	Blob1…
345345	Blob2…

Horizontal partitioning

Customers

SSN	Name	City	Country
234234	Mary	Houston	USA
345345	Sue	Seattle	USA
345343	Joan	Seattle	USA
234234	Ann	Portland	USA
--	Frank	Calgary	Canada
--	Jean	Montreal	Canada

CustomersInHouston

SSN	Name	City	Country
234234	Mary	Houston	USA

CustomersInSeattle

SSN	Name	City	Country
345345	Sue	Seattle	USA
345343	Joan	Seattle	USA

CustomersInCanada

SSN	Name	City	Country
--	Frank	Calgary	Canada
--	Jean	Montreal	Canada

I/O-based horizontal partitioning

- Reduce the time required to retrieve relations from disk
  - Spread relations to multiple disks
- Partitioning techniques (number of disks = n):

  Round-robin:

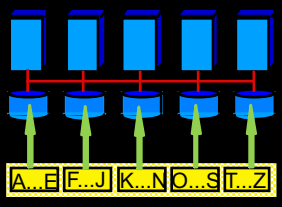
  Send the i^{th} record to disk $i \bmod n$

  Hash partitioning:
  - Choose one or more attributes as the partitioning attributes, e.g., $<a, b>$
  - Choose hash function h such that $0 \le h(a_i, b_i) \le n - 1$
  - Send record i to disk $h(a_i, b_i)$

  Range partitioning:
  - Choose a partitioning attribute v and a partitioning vector $[v_0, v_1, ..., v_{n-2}]$
  - Records with
    - $v < v_0$ go to disk 0
    - $v_i \le v < v_{i+1}$ go to disk $i+1$
    - $v_{n-2} \le v$ go to disk $n-1$

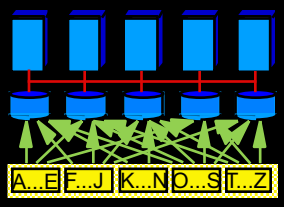
Partitioned data storage

Range



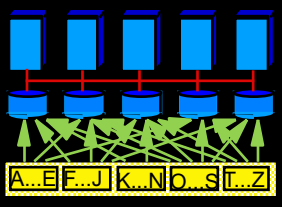
Good for point queries, equijoins, range queries group-by

Hash



Good for point queries, equijoins

Round Robin



Good to spread load

Shared disk/memory database systems less sensitive to partitioning
Shared nothing database systems benefit from "good" partitioning

PostgreSQL declarative partitioning

- Available starting with PostgreSQL 10
- Partitioning methods: list, range, (WIP: hash)
- Partition key: single or multiple columns, expressions
- Sub-partitioning
  - Partitioned partitions
  - Mixed sub-partitioning
- Development efforts
  - Several previous attempts by many people
  - First patch proposed in August 2015
  - Got committed in Dec 2016, bug fixes, doc changes continue ...

Example 1 – PostgreSQL 10 declarative syntax

```
CREATE TABLE numbers (x INTEGER)
PARTITION BY RANGE (x);


CREATE TABLE negatives
PARTITION OF numbers
FOR VALUES FROM (MINVALUE) TO (0);


CREATE TABLE positives
PARTITION OF numbers
FOR VALUES FROM (0) TO (MAXVALUE);
```

Example 1 – numbers relation description

```
db=> \d+ numbers
                                  Table "public.numbers"
 Column |  Type   | Collation | Nullable | Default | Storage | Stats target | Description
--------+---------+-----------+----------+---------+---------+--------------+-------------
 x      | integer |           |          |         | plain   |              |
Partition key: RANGE (x)
Partitions: negatives FOR VALUES FROM (MINVALUE) TO (0),
            positives FOR VALUES FROM (0) TO (MAXVALUE)
```


Example 1 – negatives / positives relation description

```
db=> \d+ negatives
                                 Table "public.negatives"
 Column |  Type   | Collation | Nullable | Default | Storage | Stats target | Description
--------+---------+-----------+----------+---------+---------+--------------+-------------
 x      | integer |           |          |         | plain   |              |
Partition of: numbers FOR VALUES FROM (MINVALUE) TO (0)
Partition constraint: ((x IS NOT NULL) AND (x < 0))


db=> \d+ positives
                                 Table "public.positives"
 Column |  Type   | Collation | Nullable | Default | Storage | Stats target | Description
--------+---------+-----------+----------+---------+---------+--------------+-------------
 x      | integer |           |          |         | plain   |              |
Partition of: numbers FOR VALUES FROM (0) TO (MAXVALUE)
Partition constraint: ((x IS NOT NULL) AND (x >= 0))
```

Example 1 – tuple routing

```
INSERT INTO numbers
VALUES (-4), (-1), (7), (12);

SELECT * FROM numbers;
x
----
-4
-1
7
12
```

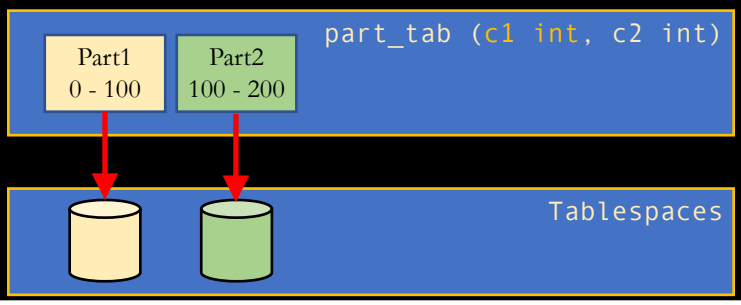
```
SELECT * FROM negatives;
x
----
-4
-1


SELECT * FROM positives;
x
----
7
12
```


Example 2 – define partition schema

```
CREATE TABLE part_tab (c1 int, c2 int)
PARTITION BY RANGE (c1);
```

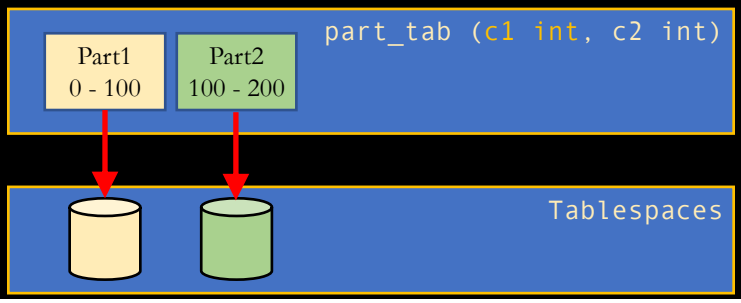
Example 2 – create partitions

```
CREATE TABLE part_tab (c1 int, c2 int)
PARTITION BY RANGE (c1);

CREATE TABLE part1
PARTITION OF part_tab FOR VALUES FROM (0) TO (100);

CREATE TABLE part2
PARTITION OF part_tab FOR VALUES FROM (100) TO (200);
```



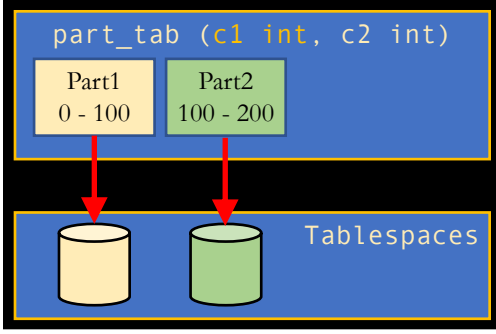
Example 2 – relation description

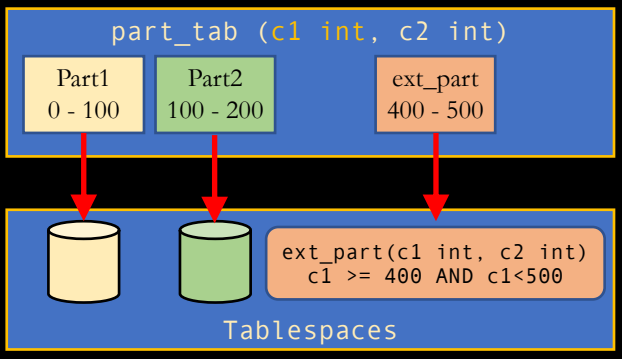
```
db=> \d+ part_tab
                                Table "public.part_tab"
 Column |  Type   | Collation | Nullable | Default | Storage | Stats target | Description
--------+---------+-----------+----------+---------+---------+--------------+-------------
 c1     | integer |           |          |         | plain   |              |
 c2     | integer |           |          |         | plain   |              |
Partition key: RANGE (c1)
Partitions: part1 FOR VALUES FROM (0) TO (100),
            part2 FOR VALUES FROM (100) TO (200)
```



Example 2 – ATTACH partition

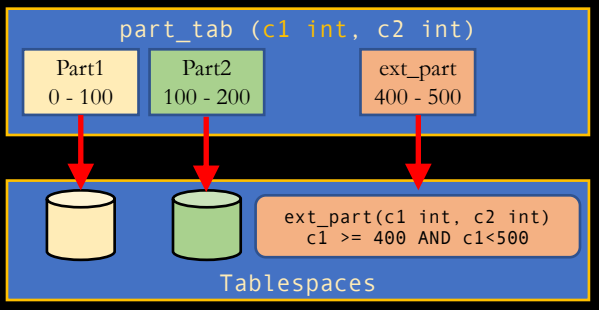
```
CREATE TABLE ext_part(c1 int, c2 int);

ALTER TABLE part_tab
ATTACH PARTITION ext_part
FOR VALUES FROM (400) to (500);
```





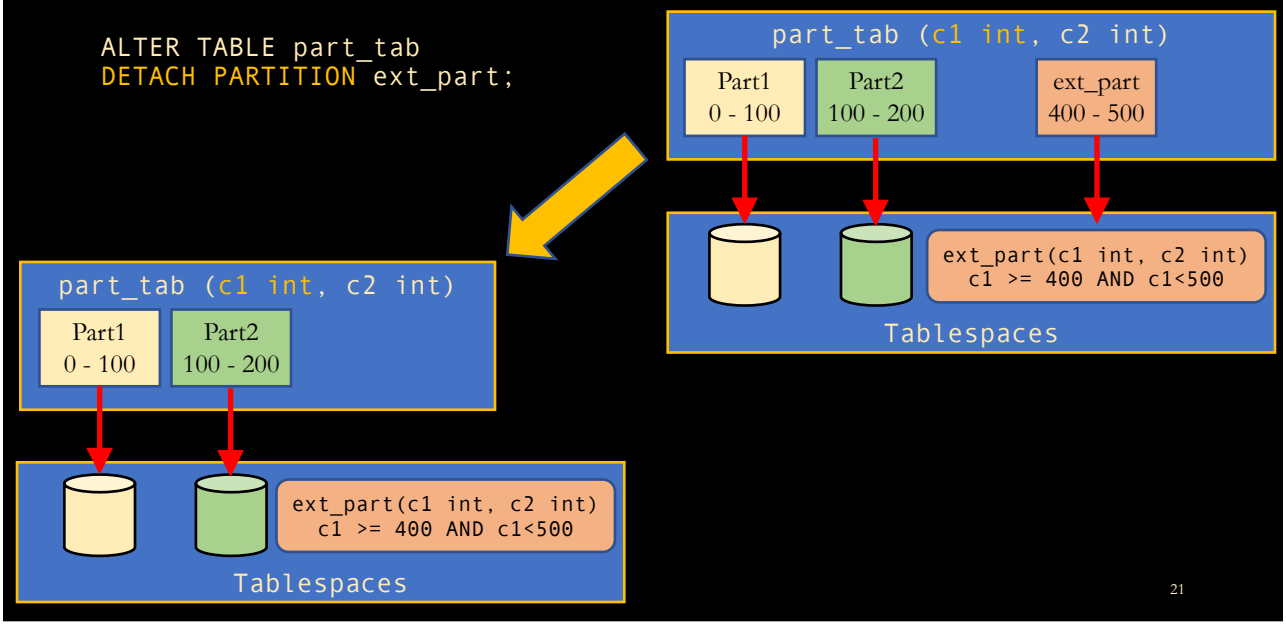
Example 2 – describe relations

```
db=> \d+ part_tab
                                Table "public.part_tab"
 Column |  Type   | Collation | Nullable | Default | Storage | Stats target | Description
--------+---------+-----------+----------+---------+---------+--------------+-------------
 c1     | integer |           |          |         | plain   |              |
 c2     | integer |           |          |         | plain   |              |
Partition key: RANGE (c1)
Partitions: ext_part FOR VALUES FROM (400) TO (500),
            part1 FOR VALUES FROM (0) TO (100),
            part2 FOR VALUES FROM (100) TO (200)
```



Example 2 – DETACH Partition

```
ALTER TABLE part_tab
DETACH PARTITION ext_part;
```



Partitions are tables

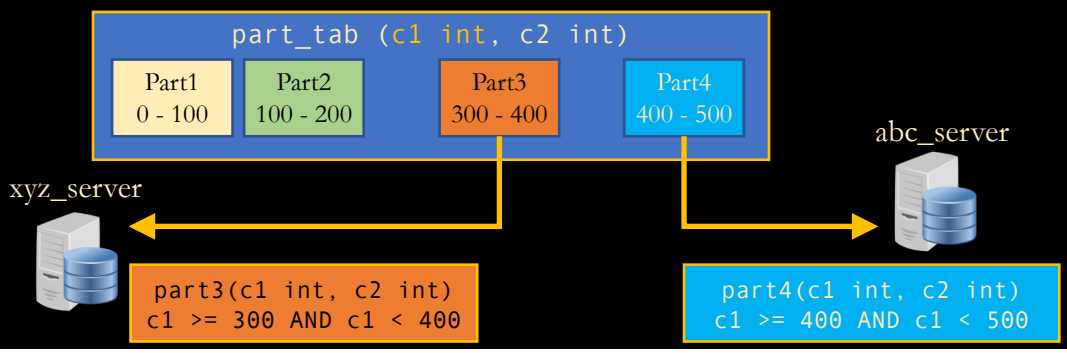
- Same columns as parent table
- Partition specific constraints, defaults, indexes, storage parameters
- Tablespace separate from that of parent
  - "hot" and "cold" partitions
- ANALYZE, CLUSTER, etc. can run separately
  - Utilities do not block the whole table
  - Work where it is required

Sub-partitioning is possible

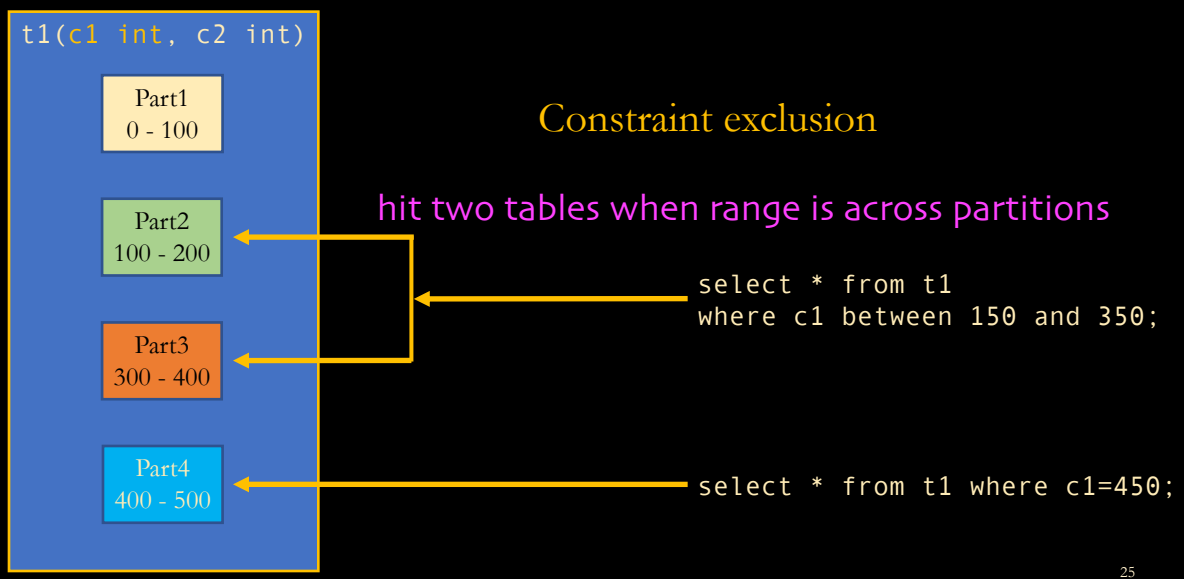
```
CREATE TABLE measurement (sense_date timestamptz, peak_temp int)
PARTITION BY RANGE (sense_date);


CREATE TABLE measurement_p1
PARTITION OF measurement
FOR VALUES FROM ('2008-02-01') TO ('2018-03-01')
PARTITION BY RANGE (peak_temp);
```


Foreign partitions

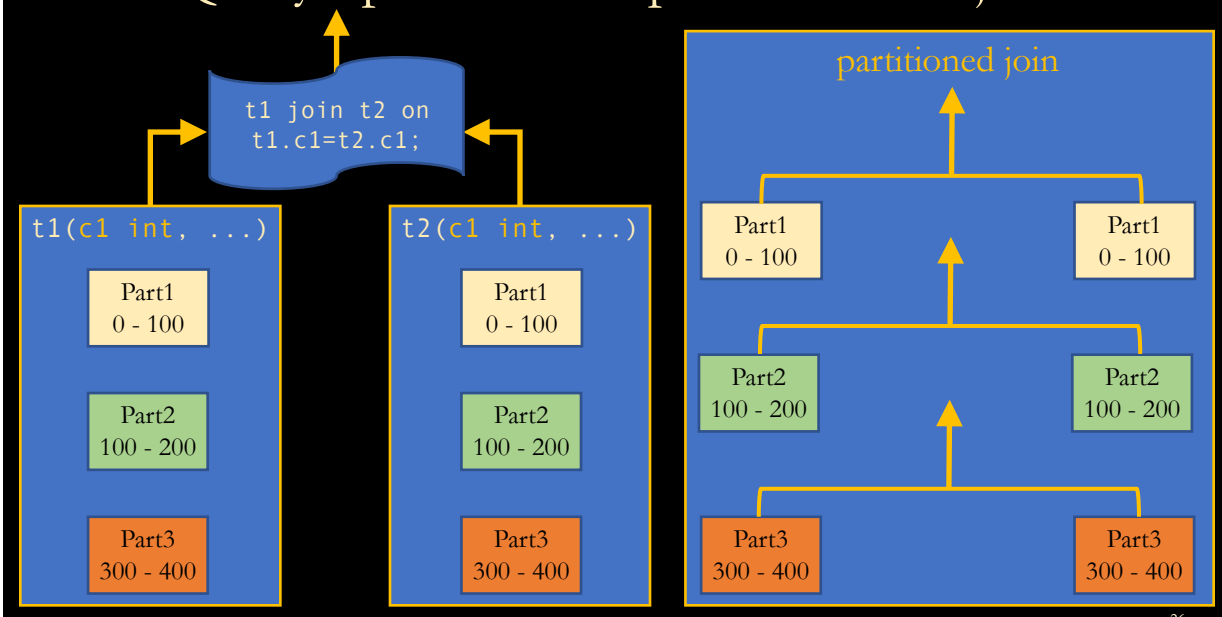
```
CREATE TABLE part_tab (c1 int, c2 int)
PARTITION BY RANGE (c1);

...

CREATE FOREIGN TABLE part3 PARTITION OF part_tab
FOR VALUES FROM (300) TO (400) SERVER xyz_server;


CREATE FOREIGN TABLE part4 PARTITION OF part_tab
FOR VALUES FROM (400) TO (500) SERVER abc_server;
```



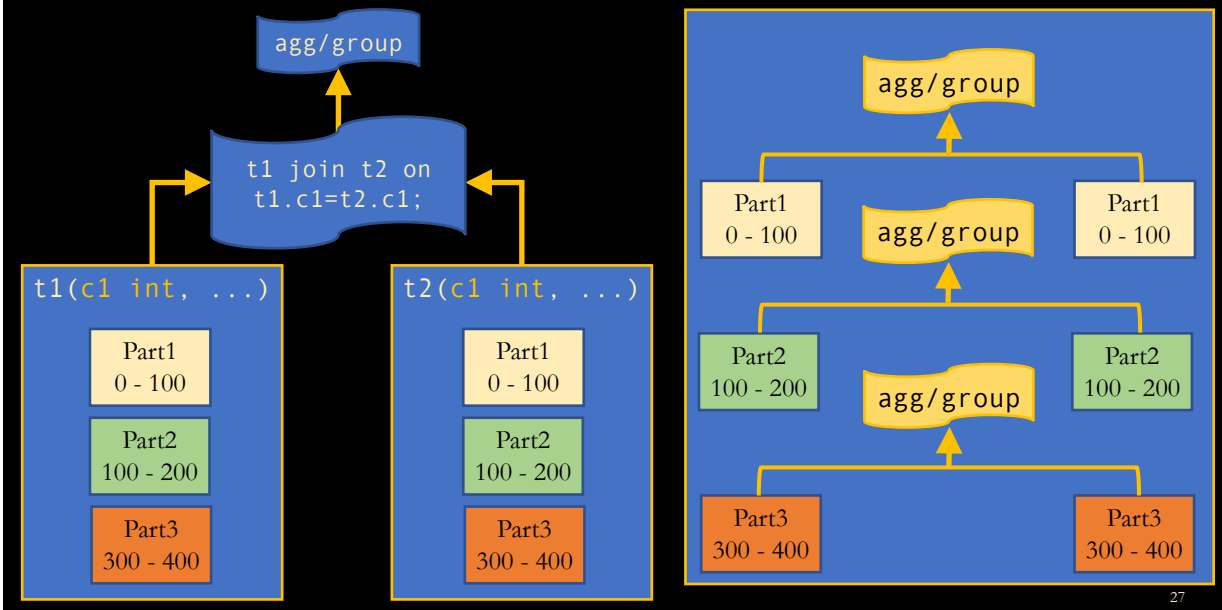
Query optimization: partition pruning



Query optimization: partition-wise join



Query optimization: partition-wise aggregation



Choosing a partitioning scheme

- Bad partitioning is worse than no partitioning
- Query optimization
  - Partition key is the key to success
  - Columns in conditions
- Storage management
  - Columns that decide the storage
  - Usually timestamp of the row
  - Easy to add and drop partitions
- Keep an eye on current limitations
  - Expected to reduce with next few releases

PostgreSQL 10 limitations

- Work has just begun …
- No triggers on partitioned table
  - Need to create those for each of the partitions
- No SPLIT, MERGE, EXCHANGE partition
  - Cannot change partitioning of data after-the-fact by "splitting" a partition or by "merging" partitions
  - Cannot move rows from one partition to another as part of executing an UPDATE that modifies the partition key
- No automatic creation of partitions for incoming data
  - No "default" partition to capture data for which no partition is defined
- No global indexes
  - No primary key, no UNIQUE constraints on partitioned tables
- No tuple routing to foreign partitions
  - No INSERTs via parent table

Relationship with inheritance

- Partitioning is really a subset of the inheritance model
  - Although it imposes more constraints on the schema design and provides more information to the system
- Currently uses the same optimizer code as used to perform inheritance planning
  - And hence suffers the same problems as inheritance when using large number of partitions (child tables)
- Partitioning offers information about partitioning in a more suitable format than when using inheritance
  - Makes it possible to implement faster algorithms in the planner for partitioned tables using this information
  - Makes it possible to implement scalable algorithms
  - Makes it possible to create partition-wise plans

Partitioning with inheritance (pre-v10 PostgreSQL)

```
CREATE TABLE part_tab (c1 int, c2 int)
PARTITION BY RANGE (c1);
```

- Declarative partitioning:

```
CREATE TABLE part1
PARTITION OF part_tab FOR VALUES FROM (0) TO (100);
```

- Inheritance partitioning (works in pre-v10 PostgreSQL):

```
CREATE TABLE part1 (
    CHECK ( c1 >= 0 AND c1 < 100 )
) INHERITS (part_tab);
```


Check the PostgreSQL documentation

https://www.postgresql.org/docs/10/static/ddl-partitioning.html

Comparison of partitioning techniques

- Evaluate how well partitioning techniques support the following types of data access:
  1. Scanning the entire relation
  2. Locating a tuple associatively – **point queries**
     - E.g., $r.A = 25$
  3. Locating all tuples such that the value of a given attribute is within a specified range – **range queries**
     - E.g., $10 \leq r.A < 25$

Evaluating round robin partitioning

- Good for sequential scan of entire relation on each query
  - All disks have almost an equal number of tuples
  - Data retrieval work is thus well balanced across disks
- Not great for point queries
  - Tuples are scattered across all disks; need to check all of them
- Range queries are difficult to process
  - No clustering; tuples are scattered across all disks

Evaluating hash partitioning

- Good for sequential scan
  - Assuming hash function is good, and partitioning attributes form a key, tuples will be equally distributed between disks
  - Retrieval work is then well balanced between disks
- Good for point queries on partitioning attribute
  - Can lookup single disk, leaving others available for answering other queries
  - Index on partitioning attribute can be local to disk, making lookup and update more efficient
- No clustering, so difficult to answer range queries

Evaluating range partitioning

- Good for sequential scan
- Good for point queries on partitioning attribute
  - Only one disk needs to be accessed
- Good for range queries on partitioning attribute
  - Provides data clustering by partitioning attribute value
  - One or a few disks may need to be accessed; rest available for other queries
  - Good if result tuples are from one or a few partitions
  - If many blocks are to be fetched, they are fetched from one or a few disks, potential disk access parallelism is wasted
    - Example of execution skew

Partitioning a relation across disks

- If a relation contains only a few tuples which will fit into a single disk block, then assign the relation to a single disk
- Large relations are preferably partitioned across all the available disks
- If a relation consists of *m* disk blocks and there are *n* disks available in the system, then the relation should be allocated **min**(*m, n*) disks

Handling of skew

- The distribution of tuples to disks may be **skewed**
  - Some disks have many tuples, others may have fewer tuples
- Types of skew:
  - **Attribute-value skew**
    - Some values appear in the partitioning attributes of many tuples
    - All the tuples with the same value for the partitioning attribute end up in the same partition
    - Can occur with range partitioning and hash partitioning
  - **Partition skew**
    - With range-partitioning, badly chosen partition vector may assign too many tuples to some partitions and too few to others
    - Less likely with hash-partitioning if a good hash function is chosen

Handling skew in range partitioning

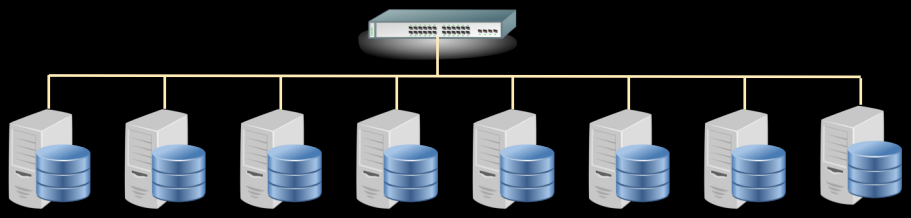
- To create a **balanced partitioning vector** (assuming partitioning attribute forms a key of the relation)
  - Sort the relation on the partitioning attribute
  - Construct the partition vector by scanning the relation in sorted order as follows
    - After every $1/n^{th}$ of the relation has been read, the value of the partitioning attribute of the next tuple is added to the partition vector
  - n denotes the number of partitions to be constructed
  - Duplicate entries or imbalances can result if duplicates are present in partitioning attributes
- Alternative technique based on **histograms** used in practice

Handling skew with virtual processor partitioning

- Skew in range partitioning can be handled elegantly using **virtual processor partitioning**
  - Create a large number of partitions
    - E.g., 10 to 20 times the number of processors
  - Assign virtual processors to partitions
    - Either in round-robin fashion or based on estimated cost of processing each virtual partition
- Basic idea:
  - If any normal partition would have been skewed, it is very likely the skew is spread over a number of virtual partitions
  - Skewed virtual partitions get spread across a number of processors, so work gets distributed evenly!

Distributing a database

- Create a "cluster" of computers connected to each other
- Each "node" in the cluster stores a fraction of the data set
- Examples:
  - Hadoop, Cassandra, MongoDB, Google File System (Colossus, BigTable), Amazon S3, Amazon DynamoDB



Challenges in distributed DBs

- Balancing storage and processing loads
- Finding data (on which node is it?)
- Analyses that combine data or process all data
  - No one node can work on all the data
- Fault tolerance
  - If we have dozens or hundreds of nodes, some are bound to fail
- Consistency
  - The different nodes cannot provide contradictory information

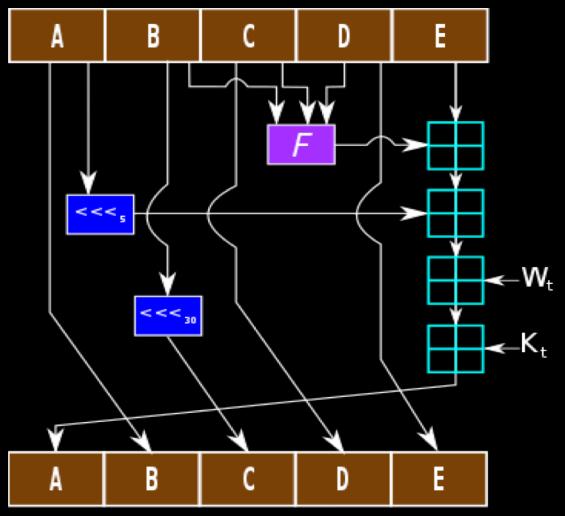
Hashing is the basis of distributed DBs

- A *hash* is an algorithm that takes a value and returns a pseudo-random value derived from it
- It's a *constant* but *unpredictable* mapping
  - A long sequence of arithmetic operations
- MD5 is a standard hash function:
  - `"Steve"` → `f6e997429bf8cb7b3b98b310a9f7ca30`
  - `"steve"` → `2666b87c682f5072f62bab0955d485ce`
  - `"Janice"` → `3837607db4754c036425cb1b2a7c8766`
  - `"1"` → `b026324c6904b2a9cb4b88d6d61c81d1`
  - `"Steve"` → `f6e997429bf8cb7b3b98b310a9f7ca30`
  - `tale_of_two_cities.txt` *(806,878 characters)* → `3ab56b74562a714a5638f94446581977`
- The same input always gives the same output
- Length of the input can vary, but output has fixed length

We can define all kinds of hash functions

- If we are dealing with text, and we want to map to a number 0-99, any of these are possible hash functions:
  - Use the length of the string (modulo 100)
  - Use the ASCII encoding of the first letter in the text (modulo 100)
  - Count the ones in the binary representation of the text (modulo 100)
  - Multiply the length of the string by the ASCII encoding of the first letter in the text and then subtract the ASCII encoding of the last letter in the text (modulo 100)
- These all produce a number in the range 0-99 and they always give the same result for a given input, but the output is not well-balanced
  - Some numbers may be output much more frequently than others
- Standard hashes like MD5 and SHA-1 are very carefully designed

SHA-1 hash function illustration



One iteration within the SHA-1 compression function:
A, B, C, D and E are 32-bit words of the state;
F is a nonlinear function that varies;
$\lll_n$ denotes a left bit rotation by *n* places;
n varies for each operation;
W_t is the expanded message word of round t;
K_t is the round constant of round t;
$\boxplus$ denotes addition modulo 2^{32}.

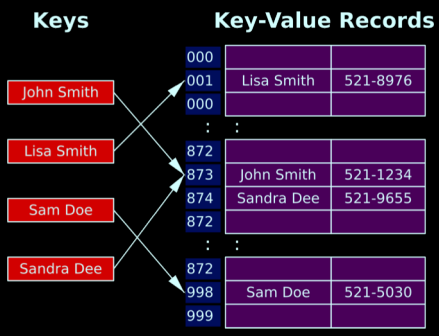
Hash table

- Stores (*key, value*) pairs
  - This abstract data type is called a *dictionary*, or *map*
- For example:
  - A word and its definition
    - "word" → "a single distinct meaningful element of speech or writing, …"
    - "hash" → "a dish of cooked meat cut into small pieces and cooked again, …"
  - A database table's primary key and the rest of the columns in the row:
    - StaffID → [StfFirstName, StfLastName, StfStreetAddress, StfCity, StfState, ...]
    - 98005 → ["Suzanne", "Viescas", "15127 NE 24th, #383", …]
    - 98007 → ["Gary", "Hallmark", "Route 2, Box 203B", …]

key *value*

Hash table mechanics

- *Hash the key* to determine the address where the value is stored

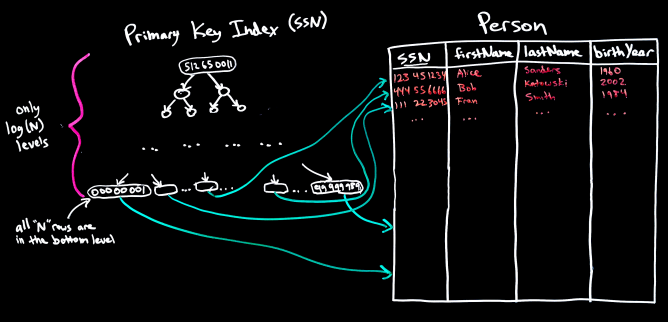


- If the address is already filled, then use the next open slot (linear probing)
  - This is called a *collision* and there are other strategies besides "linear probing"
  - "Linear probing" not to be confused with "linear hashing"

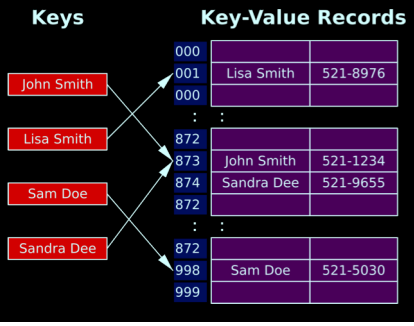
Hash indexes in SQL databases

- A hash table is an alternative to a search tree
  - It lets you find the data in one step!
  - However, compared to the tree, it can waste space
  - Also, it does not support range queries
    - Data is randomly scattered

Tree-based table index

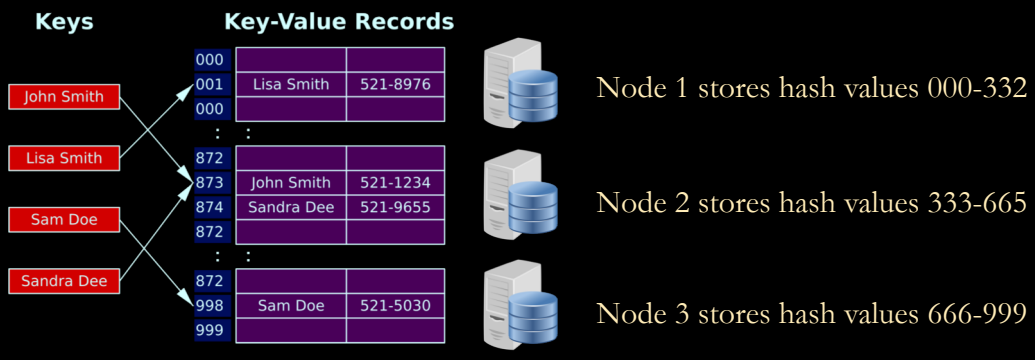


Hash-based table index



Distributed hash table

- Each cluster node is responsible for a range of hash values
- Client software can computer the key hash to determine which node to query for the data:



NoSQL databases

- NoSQL databases are distributed key-value stores
- Like one big table with just primary key
- They have a map/dictionary interface, and do not support SQL queries
  - You can only
    - *get* a value for a key
    - *put* a value for a key
  - Each operation only affects the node(s) storing that key
    - Very scalable!
- If we wanted to support full SQL and the relational model, `JOIN`s would have to pull data from many nodes in the cluster and performance would be slow

NoSQL fault tolerance

- Because there can be hundreds of nodes, NoSQL databases are designed to tolerate node failure
- Each key-value pair must be stored on multiple nodes
- However, data replication introduces *consistency* problems
  - If two nodes report different values, then which do we use?
- Common solution is to hash each key-value pair to three nodes
  - Use the "majority opinion" when reading
  - Consider a write as successful if at least two nodes succeed at storing it
- NoSQL database clusters are redundant at the software level
  - Can use inexpensive, unreliable servers because system tolerates node deaths
- SQL database servers are redundant at the hardware level
  - Use an expensive server with redundant power supplies, fans, disks, battery backup, etc.