Big Data Systems (CS4545/CS6545)
Winter 2021

Parallel Databases

Suprio Ray

University of New Brunswick, Fredericton

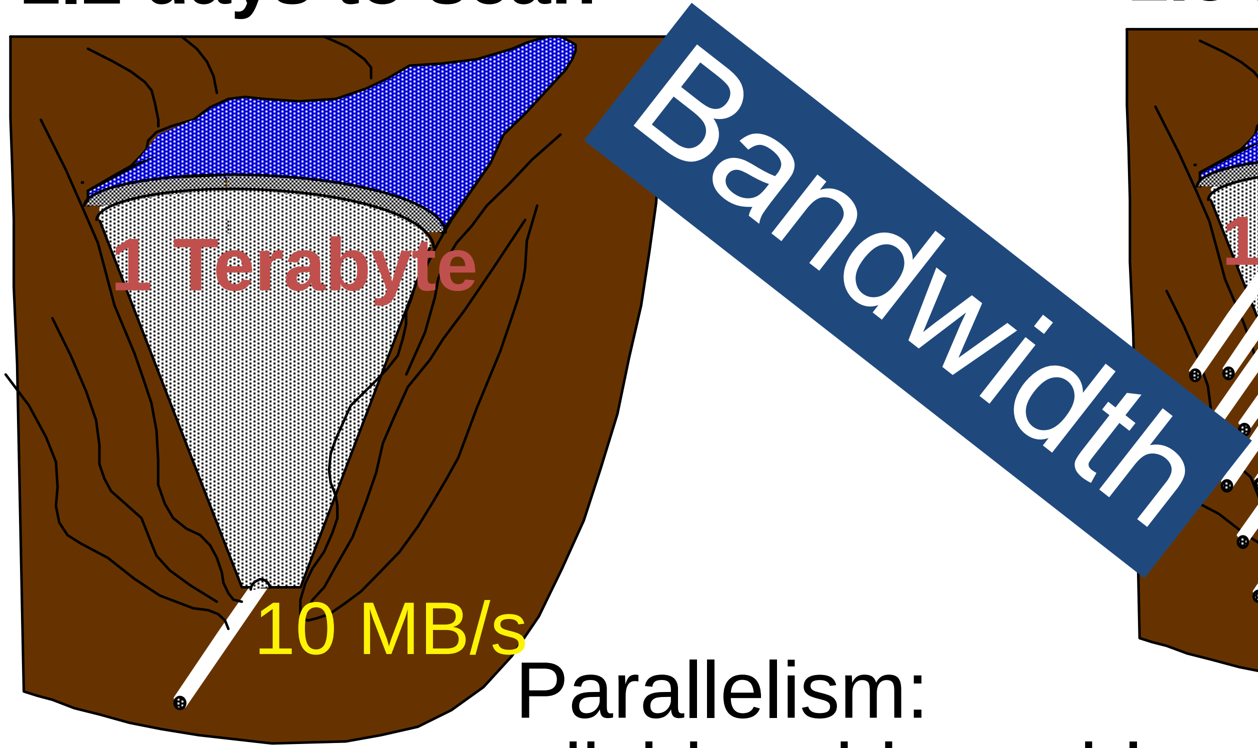
Acknowledgement

Thanks to Joe Hellerstein, Jim Gray, Y. Huang for some materials in these slides. Textbooks used include Silberschatz, Korth and Sudarshan and Taniar, Leung, Rahayu, Sushant Goel and Ramakrishnan and Gehrke.

Why Parallel Access To Data?

At 10 MB/s
1.2 days to scan

1,000 x parallel
1.5 minute to scan.



Parallelism:
divide a big problem
into many smaller ones
to be solved in parallel.

Motivation

- Parallel machines are becoming quite common and affordable
  - Prices of microprocessors, memory and disks have dropped sharply
  - Advent of multicore/manycore processors and this trend is projected to accelerate

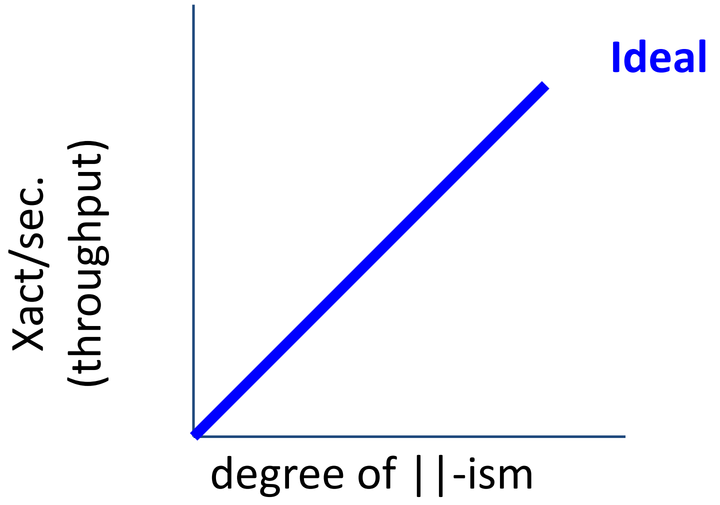
- Databases are growing increasingly large
  - large volumes of transaction data are collected and stored for later analysis
  - multimedia objects like images are increasingly stored in databases
- Increasingly, it is becoming important to support better data processing performance
  - processing time-consuming decision-support/OLAP queries
  - providing high throughput for transaction processing

Outline

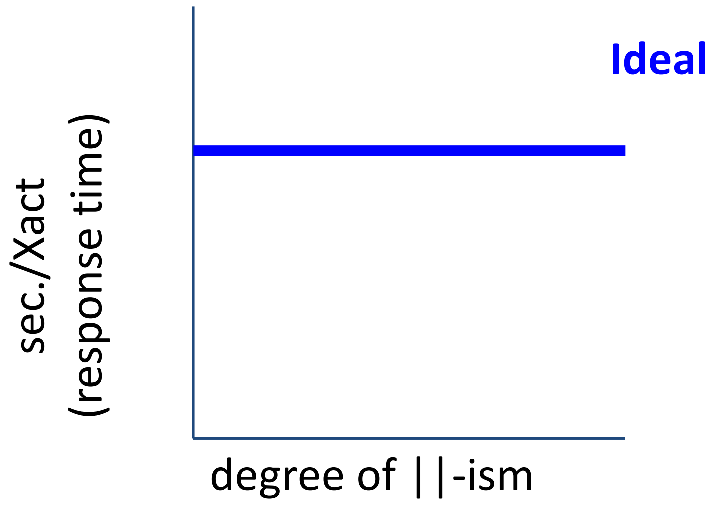
- Speedup laws
- Parallelism in DBMS and architectural issues
- Data partitioning and skew
- Parallel query processing
- Case study

Models of speedup

- Speedup
  - More resources means proportionally less time for given amount of data.

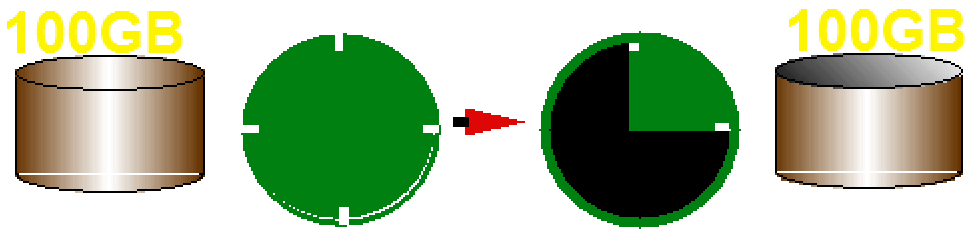


- Scaleup (scaled speedup)
  - If resources increased in proportion to increase in data size, time is constant.

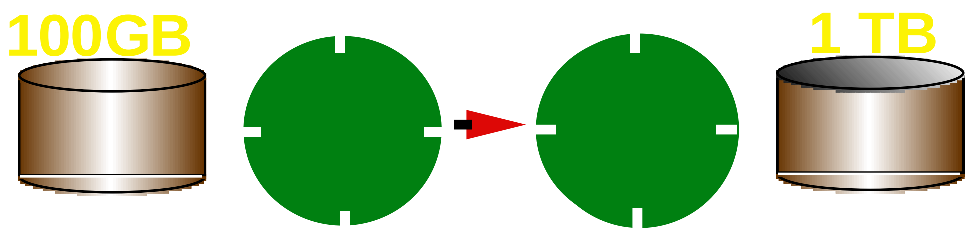


Parallelism: Speedup & Scaleup

Speedup:
- Same Job,
- More Hardware
- Less time



Scaleup:
- Bigger Job,
- More Hardware
- Same time

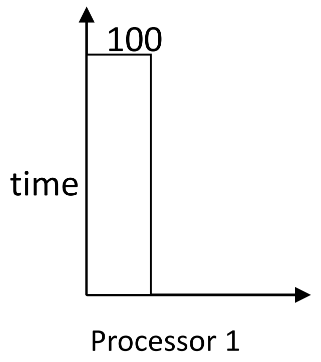


Speedup

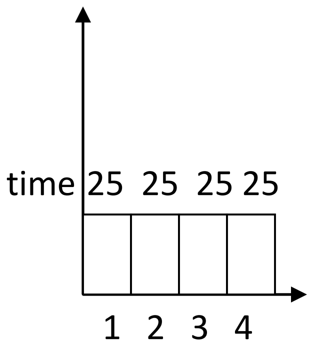
- T_1 = time for the *best* serial algorithm
- T_p= time for parallel algorithm using *p* processors

$$S_p = \frac{T_1}{T_p}$$

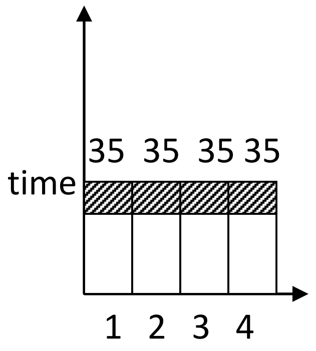
Example



(a)



(b)



(c)

$$S_p = \frac{100}{25} = 4.0,$$

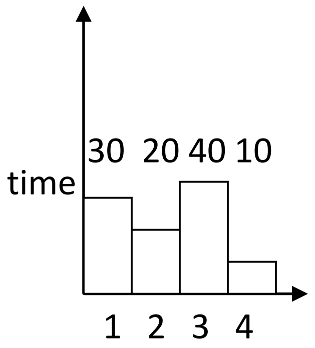
perfect parallelization

$$S_p = \frac{100}{35} = 2.85,$$

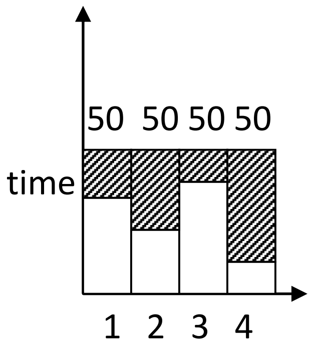
perfect load balancing

but synch cost or overhead is 10

Example (cont.)



(d)



(e)

$$S_p = \frac{100}{40} = 2.5,$$

no synch

but load imbalance

$$S_p = \frac{100}{50} = 2.0,$$

load imbalance

and synch cost

What Is “Good” Speedup?

- *Linear* speedup: best possible, normally

  $S_p = p$

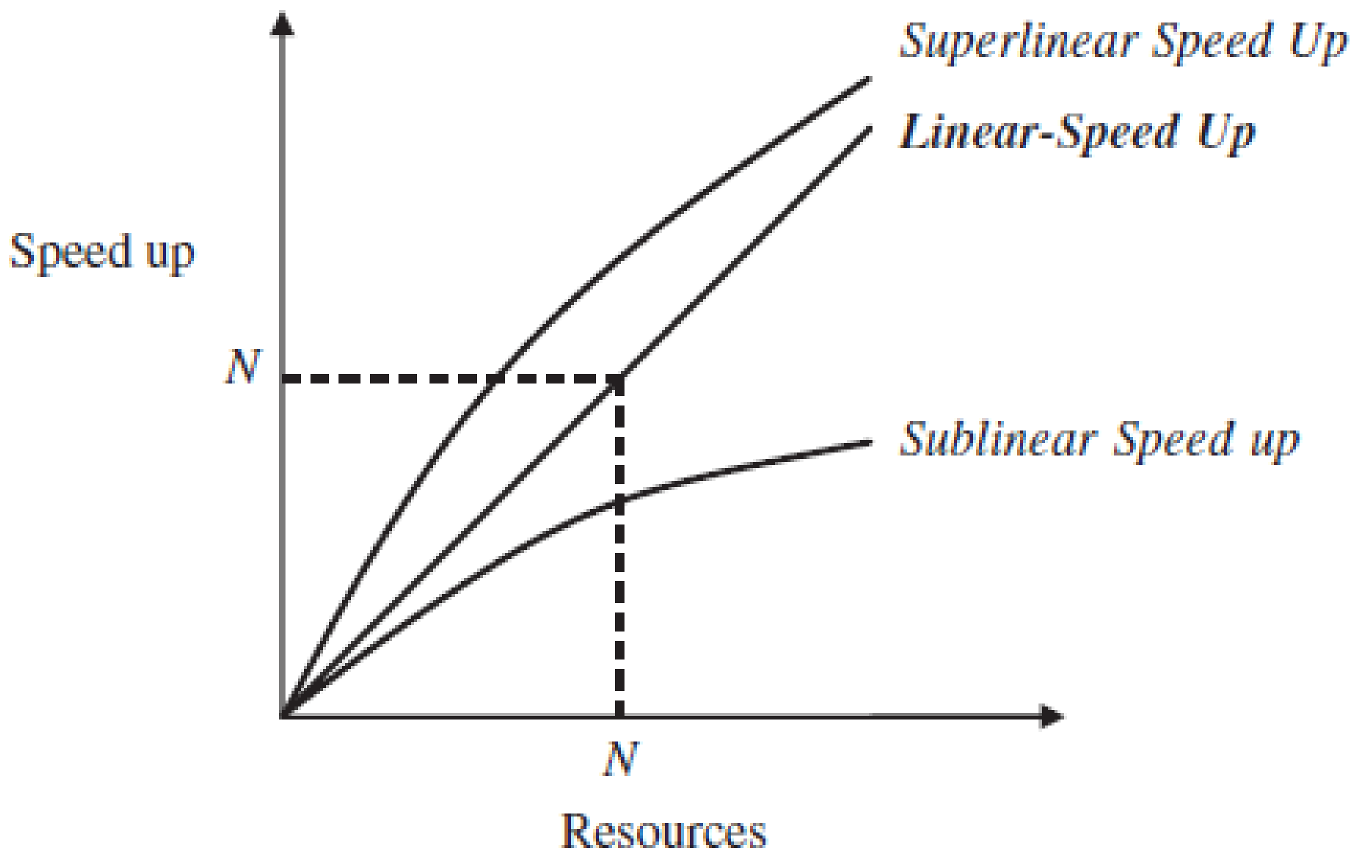
- *Superlinear* speedup: due to caching effects

  $S_p > p$

- *Sub-linear speedup:* typically achieved

  $S_p < p$

Speedup

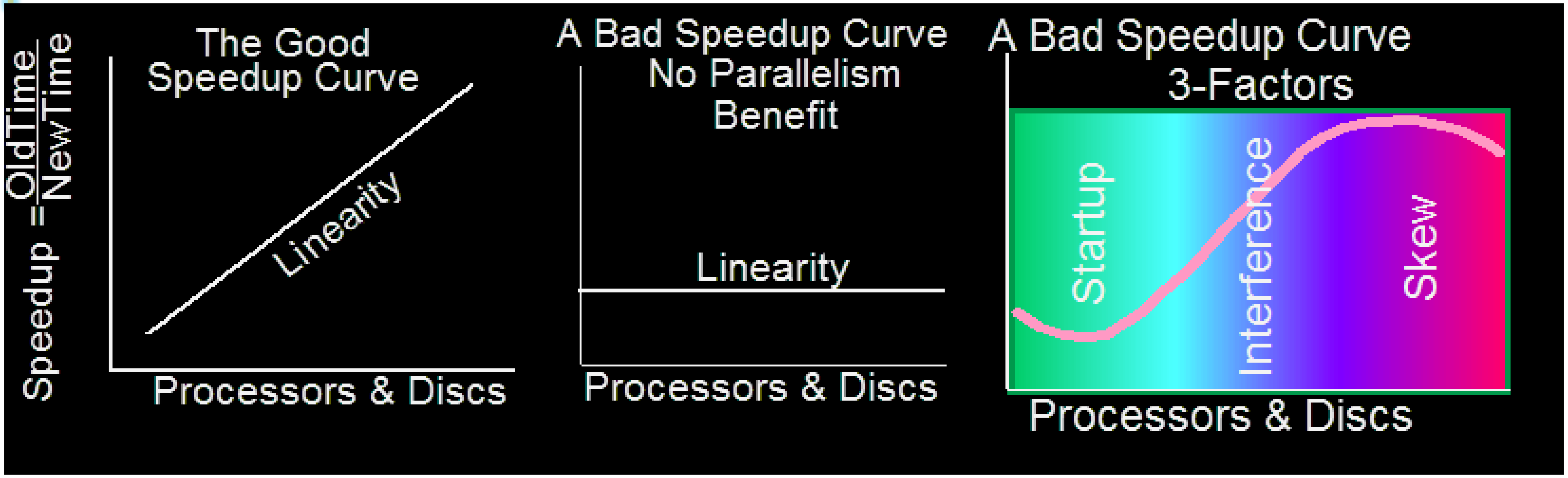

Superlinear Speed Up
Linear-Speed Up
Sublinear Speed up
Speed up
N
N
Resources

Speedup laws

- Fixed-Size Speedup (Amdahl's law)
  - Emphasis on turnaround time
  - Problem size, ***W***, is fixed

$$S_p = \frac{Uniprocessor\ \ Runtime, T_1}{Parallel\ Runtime, T_p}$$

The Perils of Parallelism



Src: Jim Gray & Gordon Bell

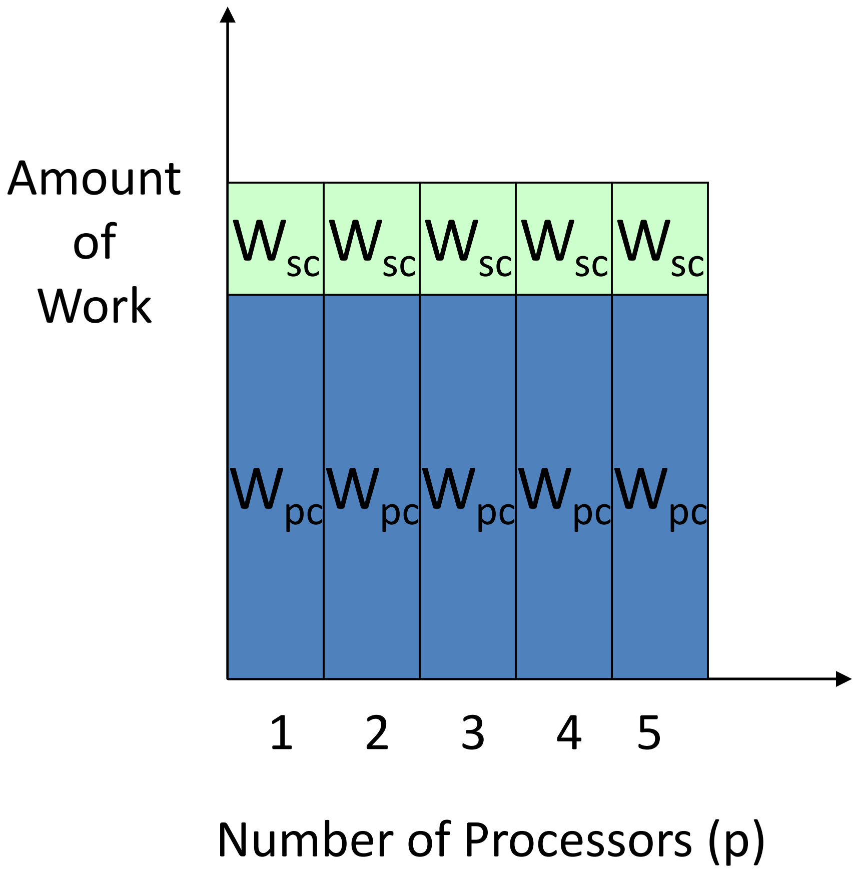
Startup: Creating processes
Opening files
Optimization

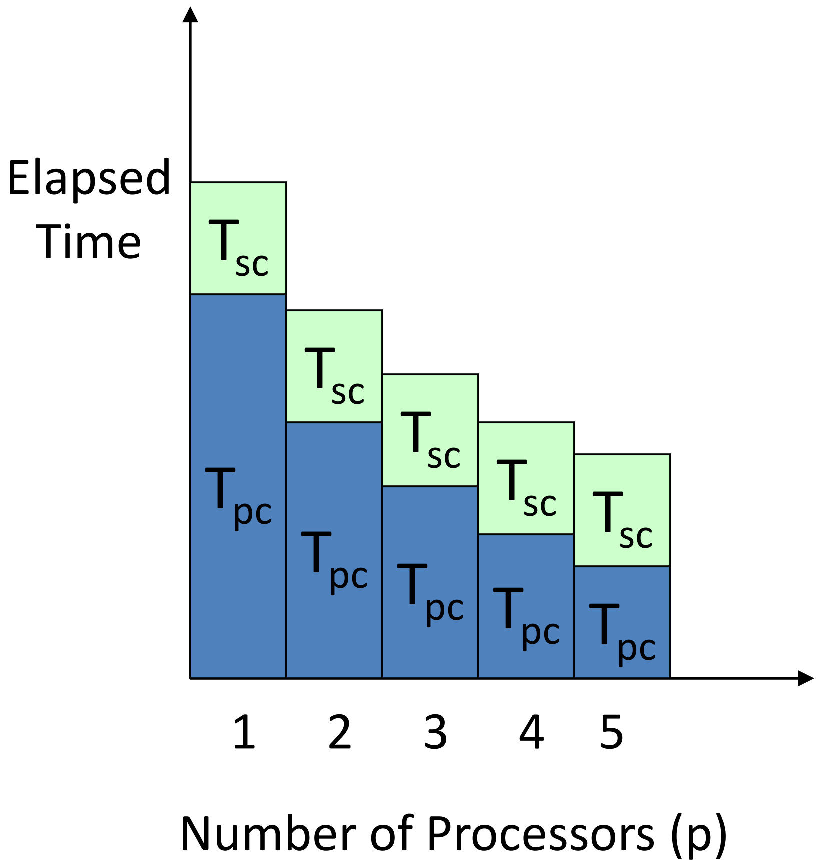
Interference: Device (cpu, disc, bus)
logical (lock, hotspot, server, log,...)

Skew: If tasks get very small, variance > service time

Speedup laws

- Fixed-Size Speedup (Amdahl Law, 67)





Here, W_{sc} = serial work component, W_{pc} is parallel work component

The amount of work is fixed

Amdahl's Law

- The performance improvement that can be gained by a parallel implementation is limited by the fraction of time parallelism can actually be used in an application

- Let α= fraction of program (algorithm) that is serial and cannot be parallelized. For instance:
  - Loop initialization
  - Reading/writing to a single disk
  - Procedure call overhead

- Parallel runtime is given by

$$T_p = (\alpha + \frac{1-\alpha}{p}) \bullet T_1$$

Amdahl's Law

- **Amdahl's law** gives a limit on speedup in terms of α

$$\text{Parallel Runtime}, T_p = T_{sc} + T_{pc}$$

$$\text{Parallel Runtime}, T_p = \alpha T_1 + \frac{(1-\alpha)T_1}{p}$$

$$Speedup, S_p = \frac{T_1}{T_p} = \frac{T_1}{\alpha T_1 + \frac{(1-\alpha)T_1}{p}} = \frac{1}{\alpha + \frac{1-\alpha}{p}}$$

Amdahl’s Law with infinite processors

- The speedup that is achievable on p processors is:

$$S_p = \frac{T_1}{T_p} = \frac{1}{\alpha + \frac{1-\alpha}{p}}$$

- If we assume that the serial fraction is fixed, then the speedup for infinite processors is limited by 1/α

$$\lim_{p->\infty} S_p = \frac{1}{\alpha}$$

- For example, if α=10%, then the maximum speedup is 10, even if we use an infinite number of processors

Parallelism: Performance is the Goal

Goal is to get 'good' performance.

Goal 1: parallel system should be
faster than serial system

Goal 2: parallel system should give
near-linear speedup

Outline

- Speedup laws
- Parallelism in DBMS and architectural issues
- Data partitioning and skew
- Parallel query processing
- Case study

Parallelism in databases

- Data can be partitioned across multiple disks for parallel I/O.

- Individual relational operations (e.g., sort, join, aggregation) can be executed in parallel
  - data can be partitioned and each processor can work independently on its own partition.

- Queries are expressed in high level language (SQL, translated to relational algebra)
  - makes parallelization easier.

- Different queries can be run in parallel with each other.
  - Concurrency control takes care of conflicts.

Thus, databases naturally lend themselves to parallelism

Inter-query parallelism

- Different queries/transactions execute in parallel with one another.

- Increase throughput

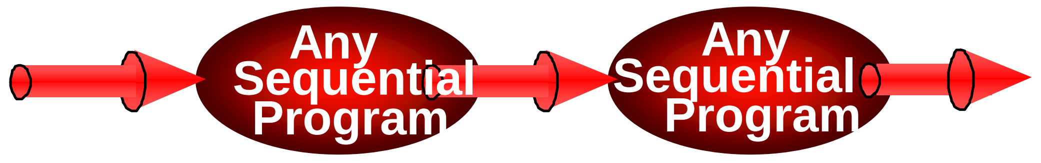
Intra-query parallelism

- Execution of a single query in parallel on multiple processors/disks
- Speed up long-running queries.
- Two complementary forms of intra-query parallelism
  - **Inter-operator Parallelism** (pipeline) – execute the different operations in a query expression in parallel.
  - **Intra-operator Parallelism** (partition) – parallelize the execution of each individual operation in the query by partitioning the dataset

Different Types of Intra-query parallelism

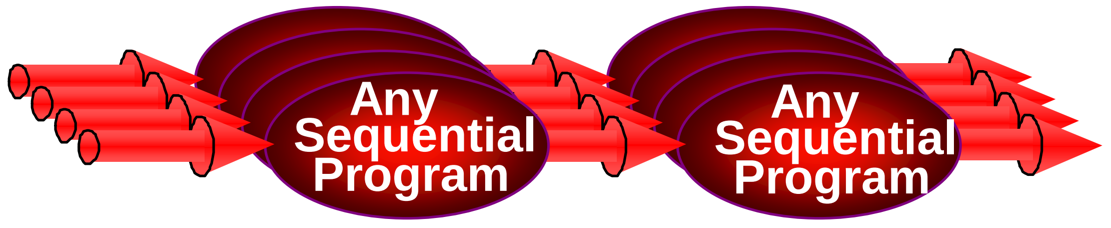
- Parallelism is natural to DBMS processing
  – *Pipeline parallelism:* many machines each doing one step in a multi-step process.

Pipeline



  – *Partition parallelism:* many machines doing the same thing to different pieces of data.

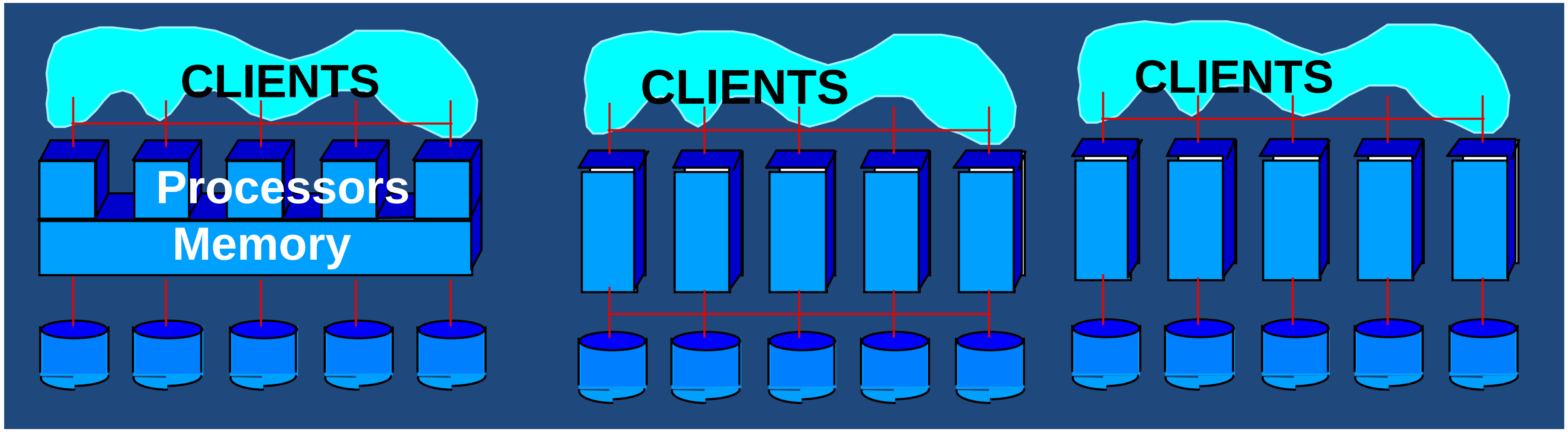
Partition



  – **Both are natural in DBMS!**

outputs split N ways, inputs merge M ways

Architecture Issue: Shared What?

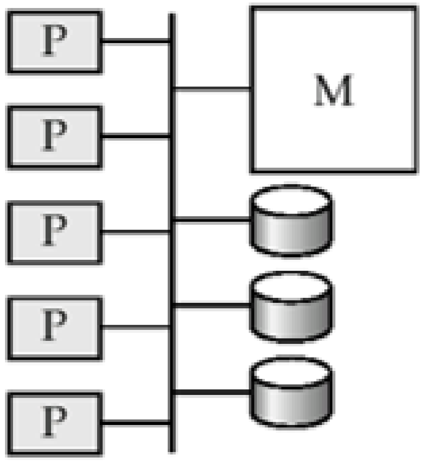


Easy to program
Expensive to build
Difficult to scaleup

Hard to program
Cheap to build
Easy to scaleup

Shared Memory

- Extremely efficient communication between processors
- Examples: Informix, Redbrick
- Downside – not scalable beyond 64 processors. Widely used for lower degrees of parallelism, although new research is changing that (w.r.t modern manycore systems)

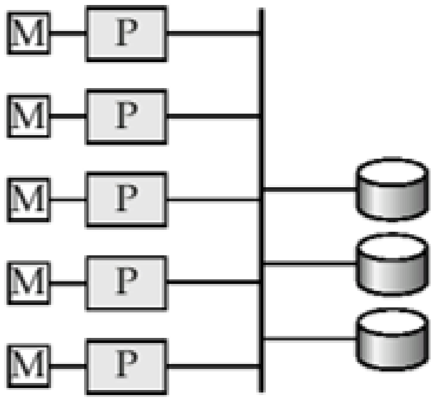


Shared Disk

- Examples: IBM Sysplex and DEC clusters (now part of HP) running Rdb (now Oracle Rdb) were early commercial users

- Downside: bottleneck at interconnection to the disk subsystem.

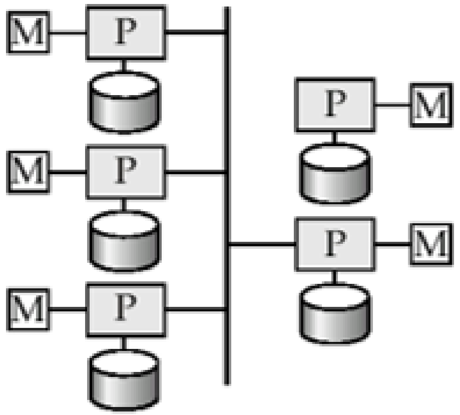
- Shared-disk systems can scale to a somewhat larger number of processors, but communication between processors is slower.



Shared Nothing

- Examples: Teradata, Tandem, Oracle-n CUBE

- Data accessed from local disks (and local memory accesses) do not pass through interconnection network, thereby minimizing the interference of resource sharing.

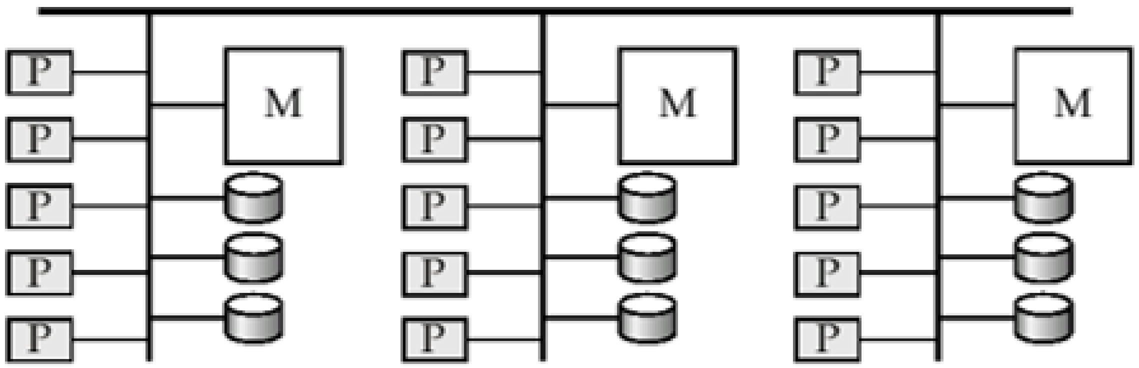


- Shared-nothing multiprocessors can be scaled up to thousands of processors without interference.

- Main drawback: cost of communication and non-local disk access; sending data involves software interaction at both ends.

Hierarchical

- Combines characteristics of shared-memory, shared-disk, and shared-nothing architectures.



Outline

- Speedup laws
- Parallelism in DBMS and architectural issues
- Data partitioning and skew
- Parallel query processing
- Case study

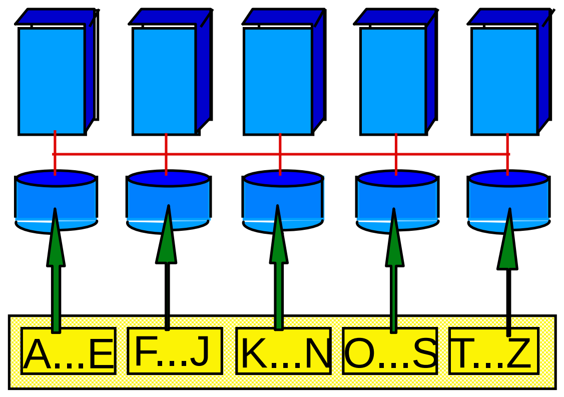
I/O parallelism with data partitioning

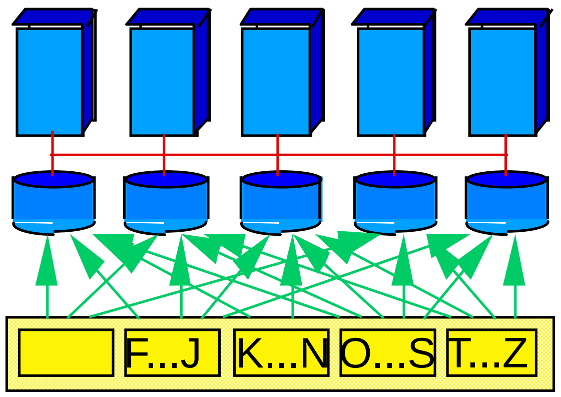
- Reduce the time required to retrieve relations from disk by partitioning the relations on multiple disks.
- Horizontal partitioning – tuples of a relation are divided among many disks such that each tuple resides on one disk
- Shared disk and memory less sensitive to partitioning, Shared nothing benefits from "good" partitioning

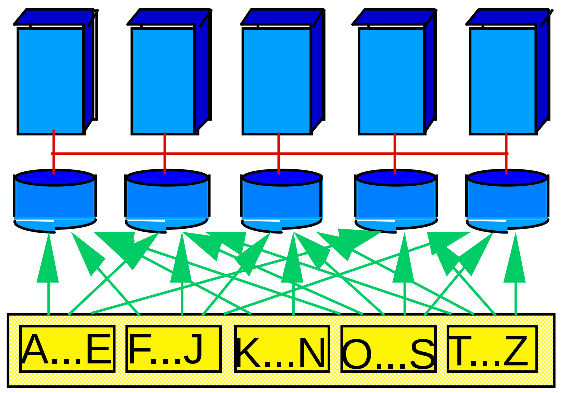
Data Partitioning

Partitioning a table:

Range **Hash** **Round Robin**







Good for equijoins, range queries group-by

Good for equijoins

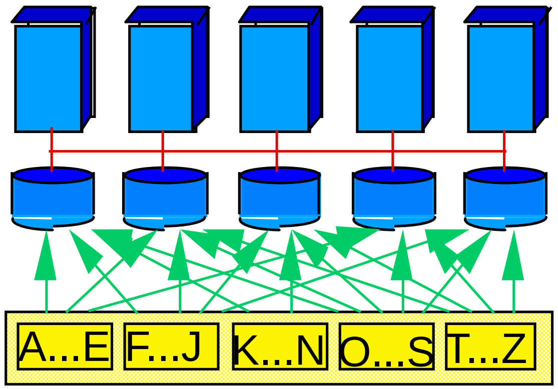
Good to spread load

Shared disk and memory less sensitive to partitioning,
Shared nothing benefits from "good" partitioning

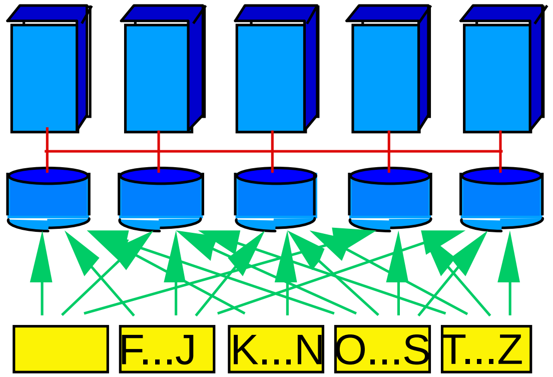
Partitioning techniques

- Partitioning techniques (number of disks = *n*):

  - **Round-robin**:
    – Send the *i*th tuple inserted in the relation to disk *i* mod *n*.

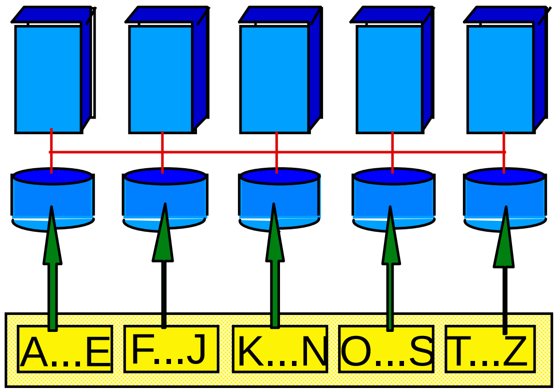


  - **Hash partitioning**:
    – Choose one or more attributes as the partitioning attributes.
    – Choose hash function *h* with range 0…*n* – 1
    – Let *i* denote result of hash function *h* applied to the partitioning attribute value of a tuple. Send tuple to disk *i*.



Partitioning techniques (contd.)

- **Range partitioning**:
  - Choose an attribute as the partitioning attribute.
  - A partitioning vector [v_o, v_1, ..., v_{n-2}] is chosen.
  - Let v be the partitioning attribute value of a tuple. Tuples such that $v \leq v_{i+1}$ go to disk $I + 1$. Tuples with $v < v_0$ go to disk 0 and tuples with $v \geq v_{n-2}$ go to disk n-1.



E.g., with a partitioning vector [5,11], a tuple with partitioning attribute value of 2 will go to disk 0, a tuple with value 8 will go to disk 1, while a tuple with value 20 will go to disk2.

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 lies within a specified range – **range queries**.
   - E.g., $10 \leq r.A < 25$.

Comparison of Partitioning Techniques (Cont.)

Round robin:

- Advantages
  - Best suited for sequential scan of entire relation on each query.
  - All disks have almost an equal number of tuples; retrieval work is thus well balanced between disks.

- Range queries are difficult to process
  - No clustering -- tuples are scattered across all disks

Comparison of Partitioning Techniques (Cont.)

Hash partitioning:

- Good for sequential access
  - 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

Comparison of Partitioning Techniques (Cont.)

Range partitioning

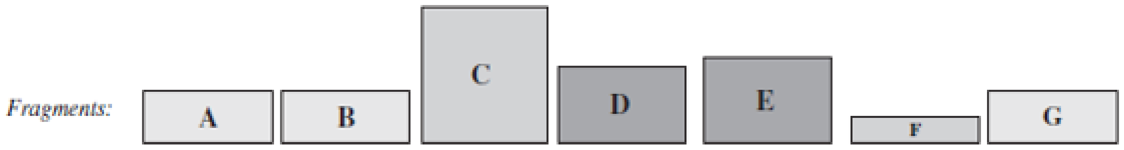
- Provides data clustering by partitioning attribute value.
- Good for sequential access
- Good for point queries on partitioning attribute: only one disk needs to be accessed.
- For range queries on partitioning attribute, one to a few disks may need to be accessed
  - Remaining disks are available for other queries.
  - Good if result tuples are from one to a few blocks.
  - If many blocks are to be fetched, they are still fetched from one to a few disks, and potential parallelism in disk access is wasted
    - Example of execution skew.

Comparison of Partitioning Techniques

	Round Robin	**Hashing**	**Range**
Sequential Scan	Best/good parallelism	Good	Good
Point Query	Difficult (why?)	Good for hash key	Good for range vector
Range Query	Difficult	Difficult	Good for range vector

Handling of Skew

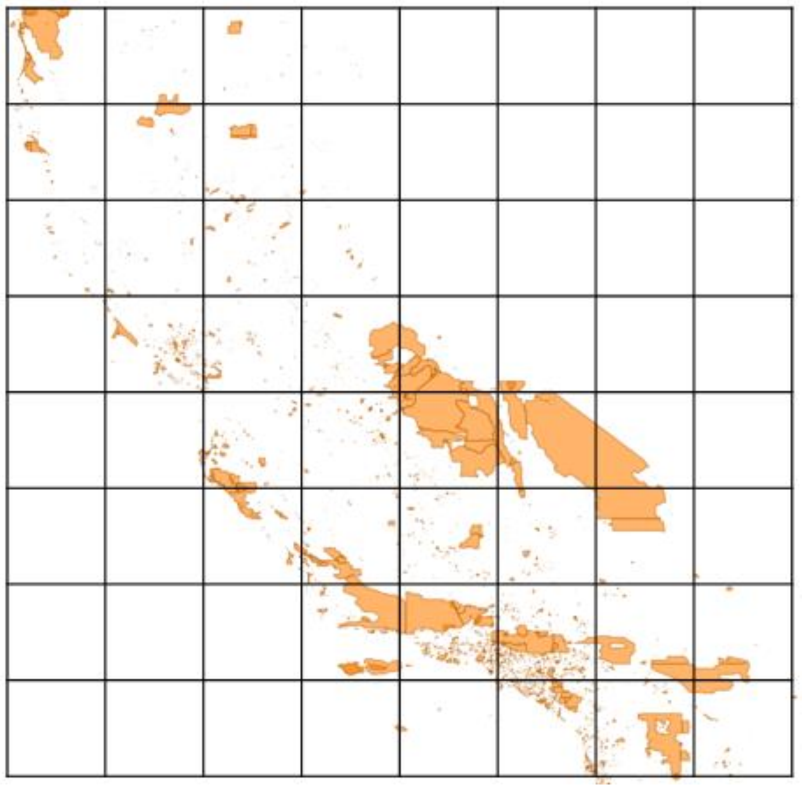
- The distribution of tuples to disks may be **skewed** — that is, some disks have many tuples, while others may have fewer tuples.



- 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.

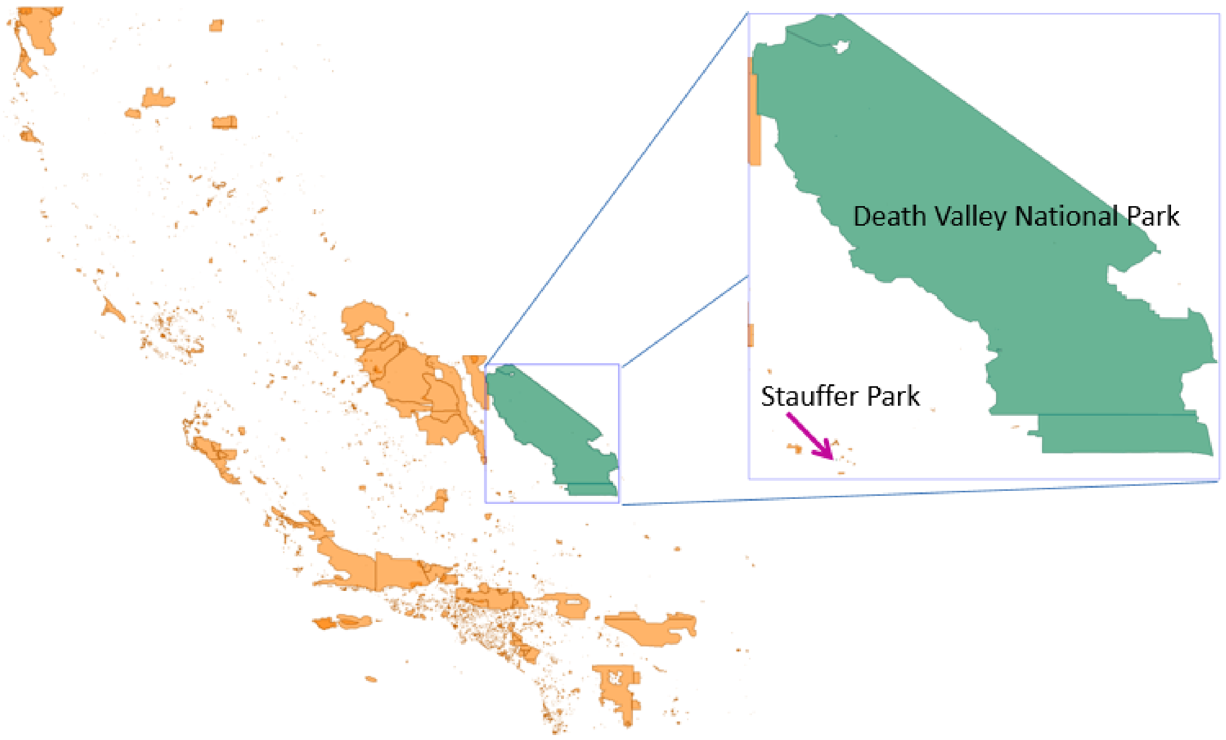
Handling of Skew

- Types of skew (continued):
  - **Execution/processing skew.**
    - Even when all partitions have the same number of tuples, some partitions may take longer than others, due to data properties



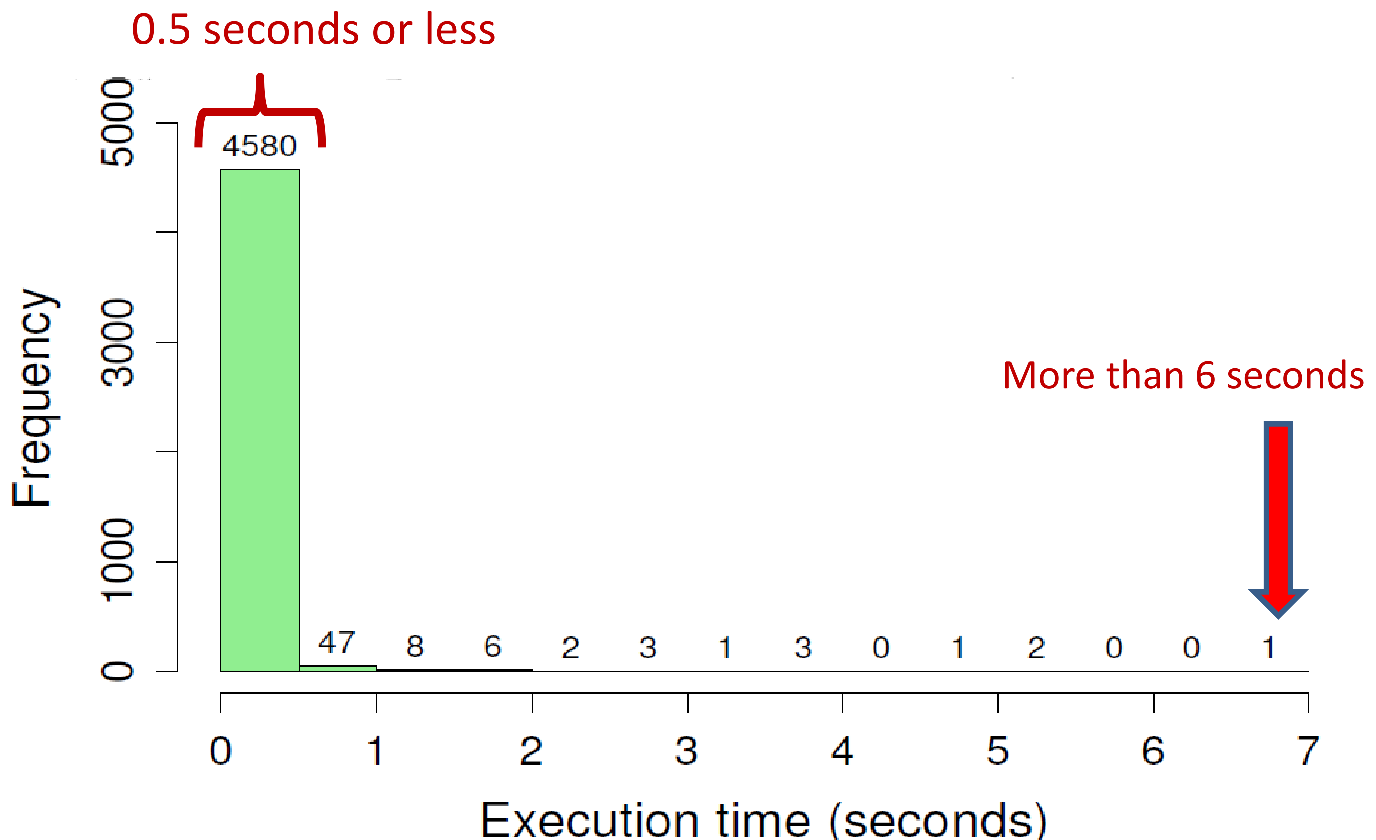
Handling of Skew

- Types of skew (continued):
  - **Execution/processing skew.**
    - Even when all partitions have the same number of tuples, some partitions may take longer than others, due to data properties



Handling of Skew

- Types of skew (continued):
  - **Execution/processing skew.**
    - Even when all partitions have the same number of tuples, some partitions may take longer than others, due to data properties

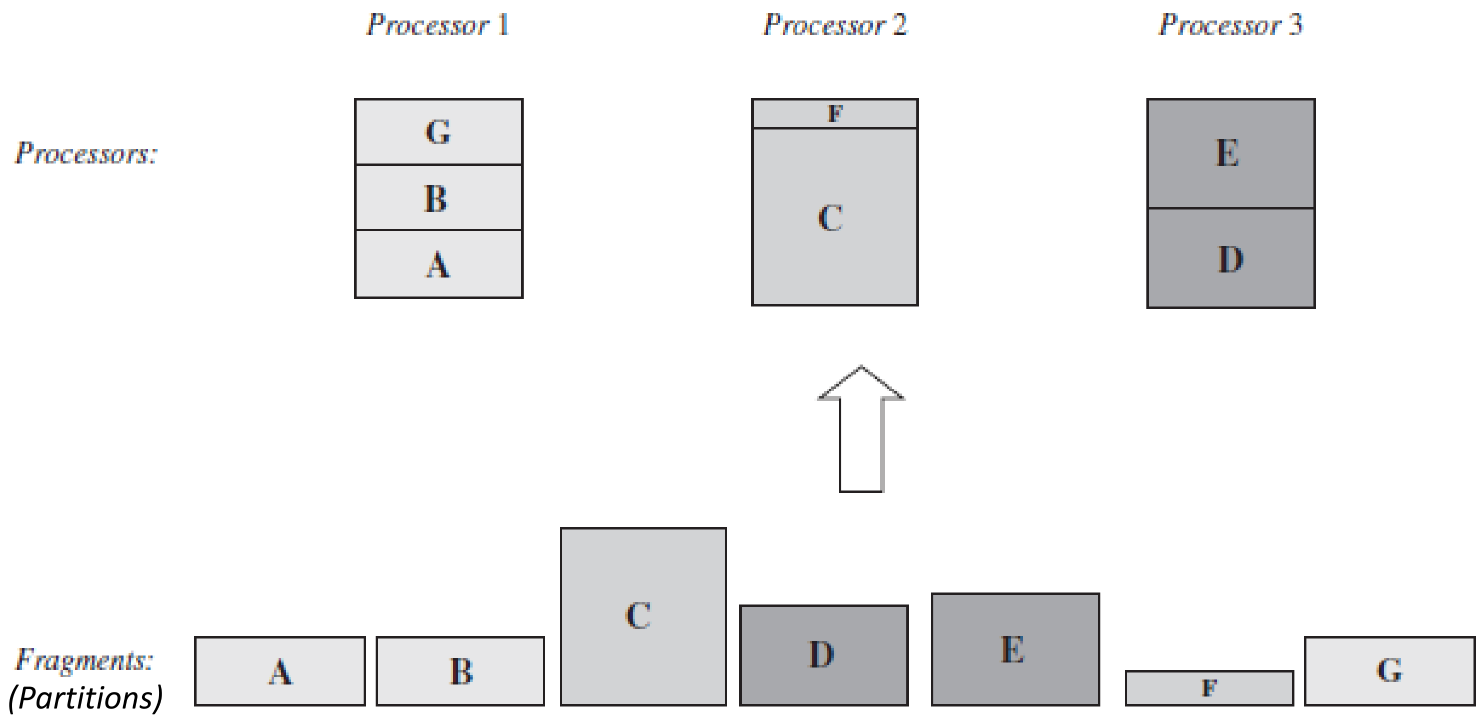


Handling Skew

- Virtual processor partitioning:
  - Create a large number of partitions (say 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!

Handling Skew

- Illustration



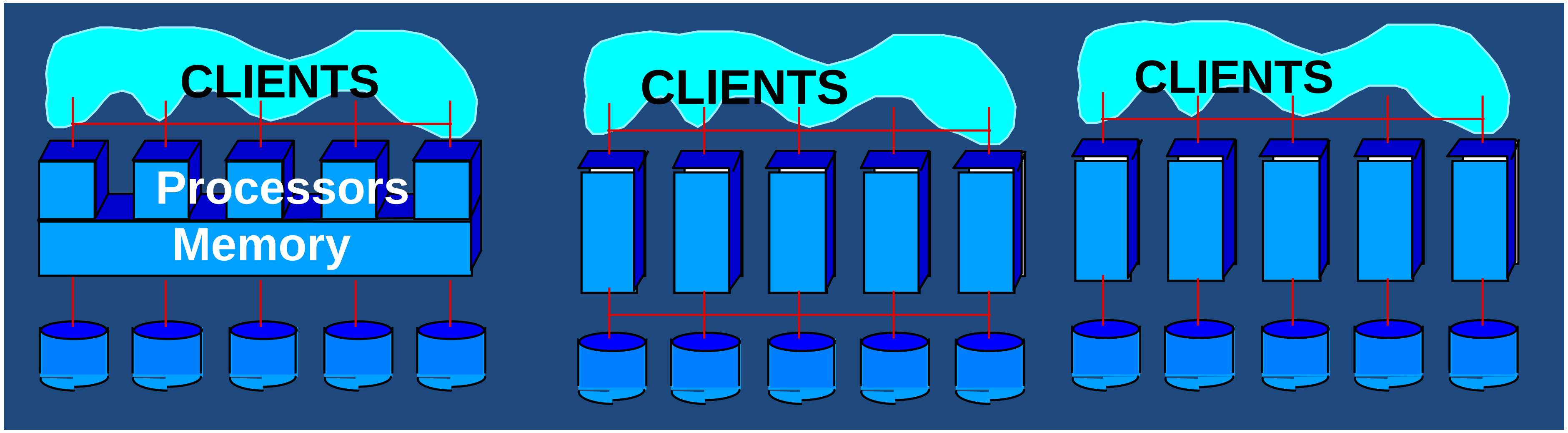
Src: Taniar, Leung, Rahayu, Sushant Goel

Outline

- Speedup laws
- Parallelism in DBMS and architectural issues
- Data partitioning and skew
- Parallel query processing
- Case study

RECALL: Parallel DB Architecture

Shared Memory (SMP) | Shared Disk | Shared Nothing (network)



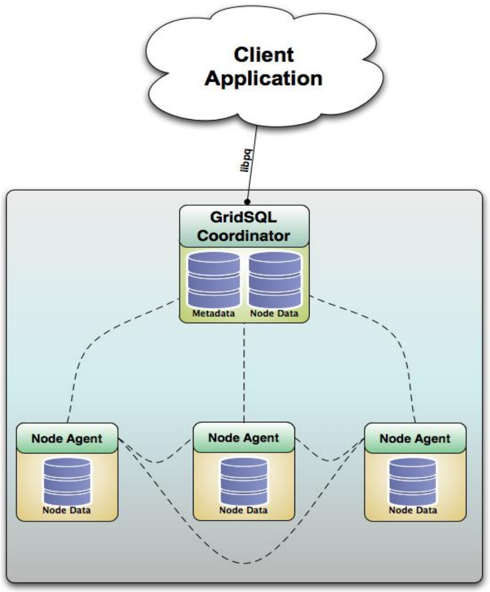
Easy to program
Expensive to build
Difficult to scaleup

Hard to program
Cheap to build
Easy to scaleup

What is GridSQL (a.k.a. Stado)?

- “Shared-Nothing”, distributed data architecture.
  - Leverage the power of multiple commodity servers while appearing as a single database to the application
- Can improve performance of PostgreSQL queries
  - Can scale linearly as more nodes are added
- Essentially...
  - Open Source

    Greenplum, Netezza or Teradata



GridSQL Details

- Designed for Parallel Querying
- Not just “Read-Only”, can execute UPDATE, DELETE
- Data Loader for parallel loading
- Standard connectivity via PostgreSQL compatible connectors: JDBC, ODBC, ADO.NET, libpq (psql)

Configuration

- Take advantage of multi-core processors
- Tables may be either replicated or partitioned
- Replicated tables for static lookup data or dimensions
- Partitioned tables for large fact tables

Partitioning

- Tables may simultaneously use GridSQL Partitioning with Constraint Exclusion partitioning

- Large queries scan a much smaller subset of data by using *subtables*

- Since each *subtable* is also partitioned across nodes, they are scanned in parallel

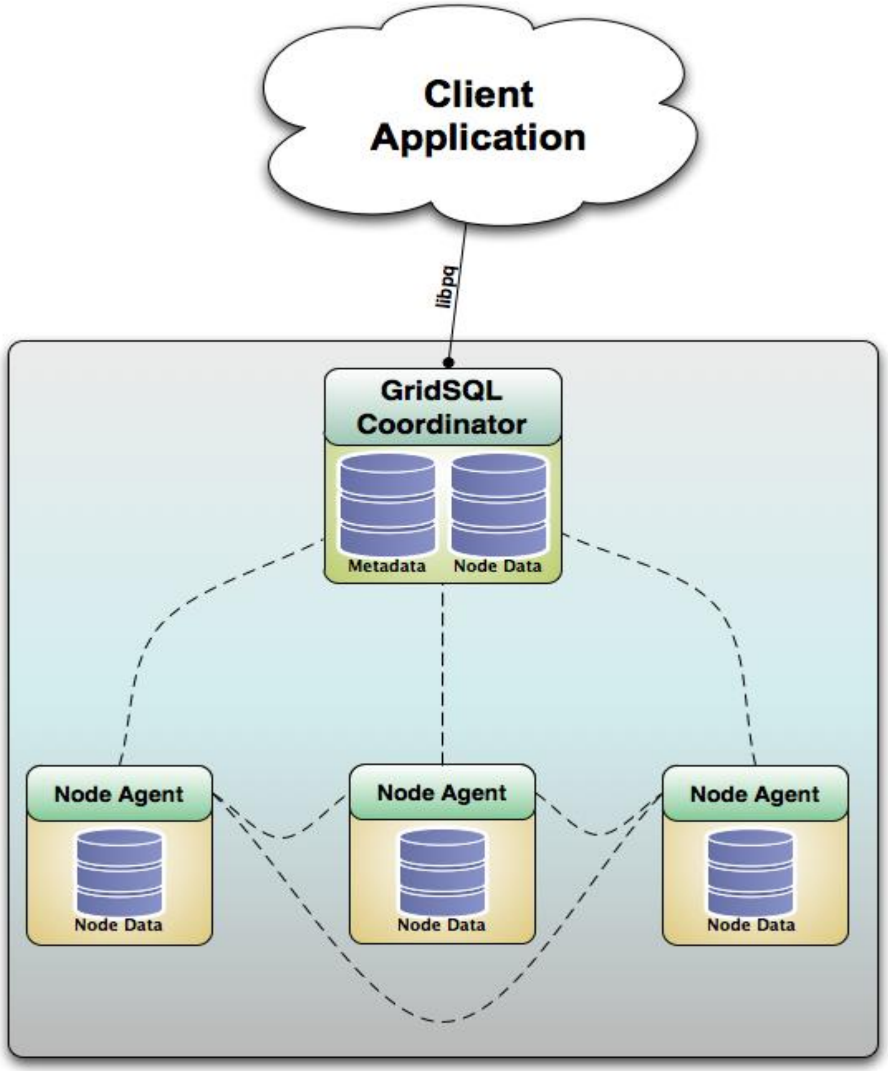
- Queries execute much faster

Architecture

- Loosely coupled,
  shared-nothing architecture

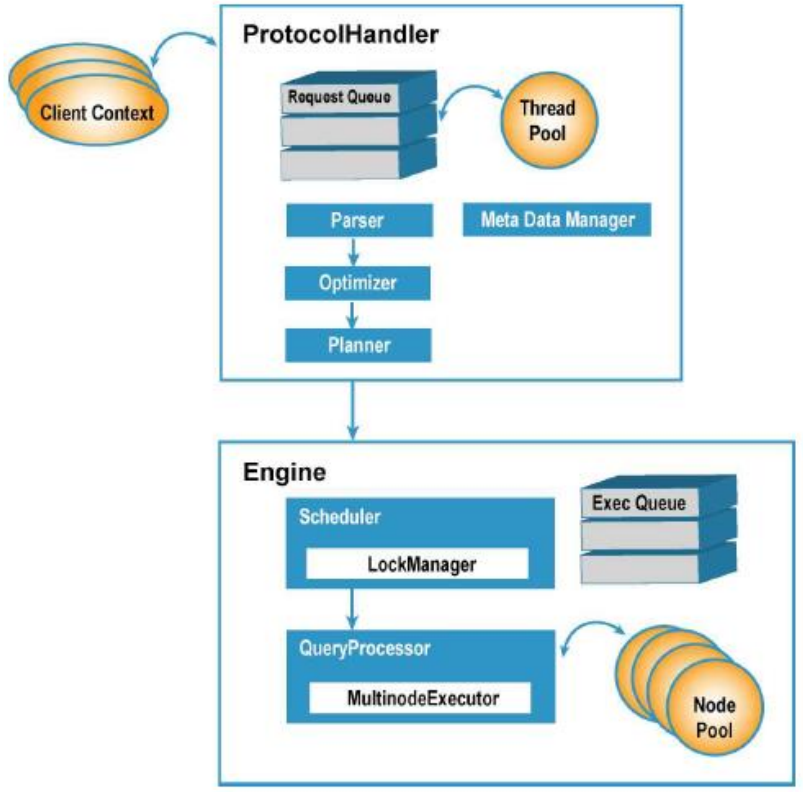
- Data repositories
  - Metadata database
  - GridSQL database

- GridSQL processes
  - Central coordinator
  - Agents



Central Coordinator

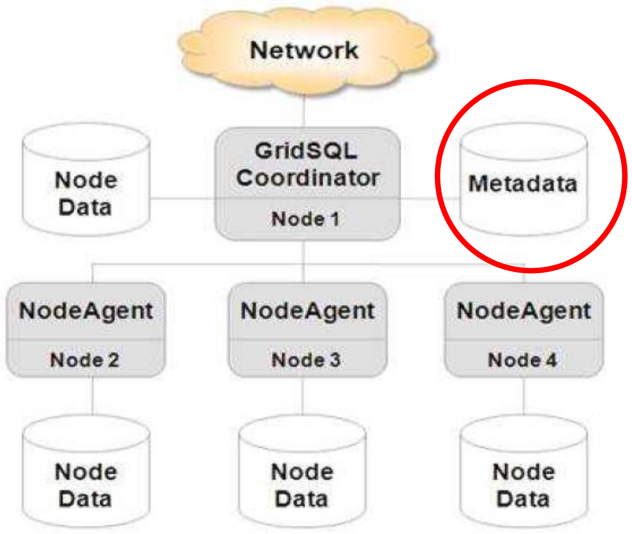
- Multi-threaded process running on designated node that manages and coordinates work between the nodes
- Makes use of metadata information
- Performs traditional DBMS functions and manages interactions with the node agents
  - Parsing and optimizing
  - Scheduling and execution



Src: EnterpriseDB

The metadata database

- Contains schema information including table partitioning and replication
- DDL issued to the GridSQL is recorded in the metadata database
- SQL requests made to the GridSQL interrogate the metadata database for partitioning and replication information to parallelize query plan
  - xsystables
  - xsyscolumns
  - xsysindexes
  - xsystabspaces
  - xsysconstraints
  - xsysindexkeys
  - xsysviews



Src: EnterpriseDB

Query Optimization

- Cost Based Optimizer
  - Takes into account Row Shipping (expensive)

- Looks for joins with *replicated tables*
  - Can be done locally
  - Looks for joins between tables on partitioned columns

Creating Tables

- Tables can be partitioned or replicated

```
CREATE TABLE region
                    (r_regionkey  INTEGER NOT NULL,
                     r_name       CHAR(25) NOT NULL,
                     r_comment  VARCHAR(152)) REPLICATED;
```

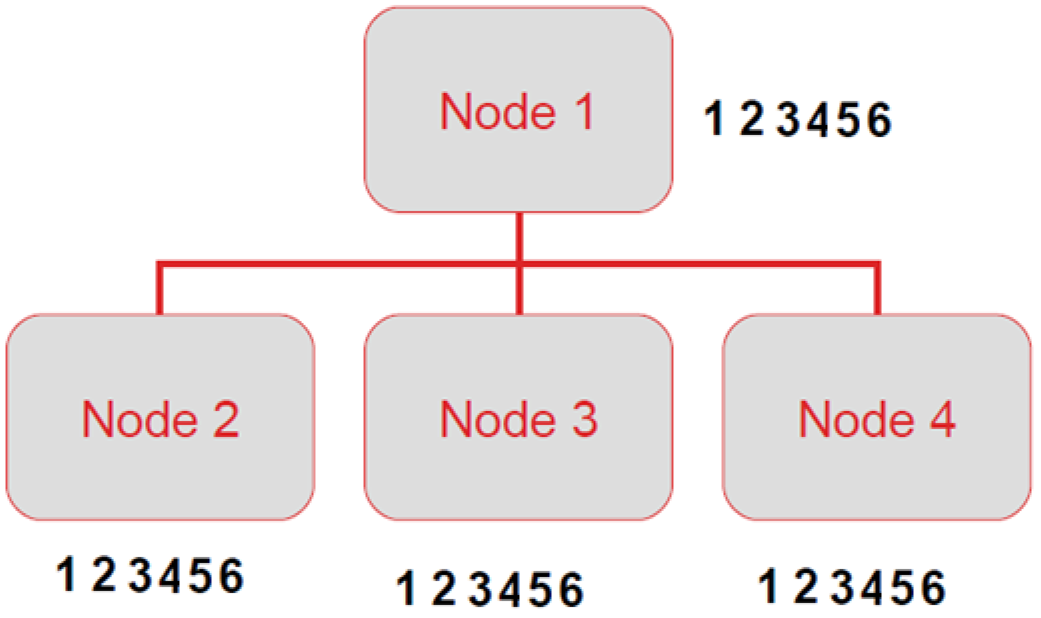
Creating Tables

- Tables can be partitioned or replicated

```
CREATE TABLE orders  (
                    o_orderkey      INTEGER NOT NULL,
                    o_custkey       INTEGER NOT NULL,
                    o_orderstatus   CHAR(1) NOT NULL,
                    o_totalprice    DECIMAL(15,2) NOT NULL,
                    o_orderdate     DATE NOT NULL,
                    o_orderpriority CHAR(15) NOT NULL,
                    o_clerk         CHAR(15) NOT NULL,
                    o_shippriority  INTEGER NOT NULL,
                    o_comment       VARCHAR(79) NOT NULL)
PARTITIONING KEY o_orderkey ON ALL;
```

Data distribution example

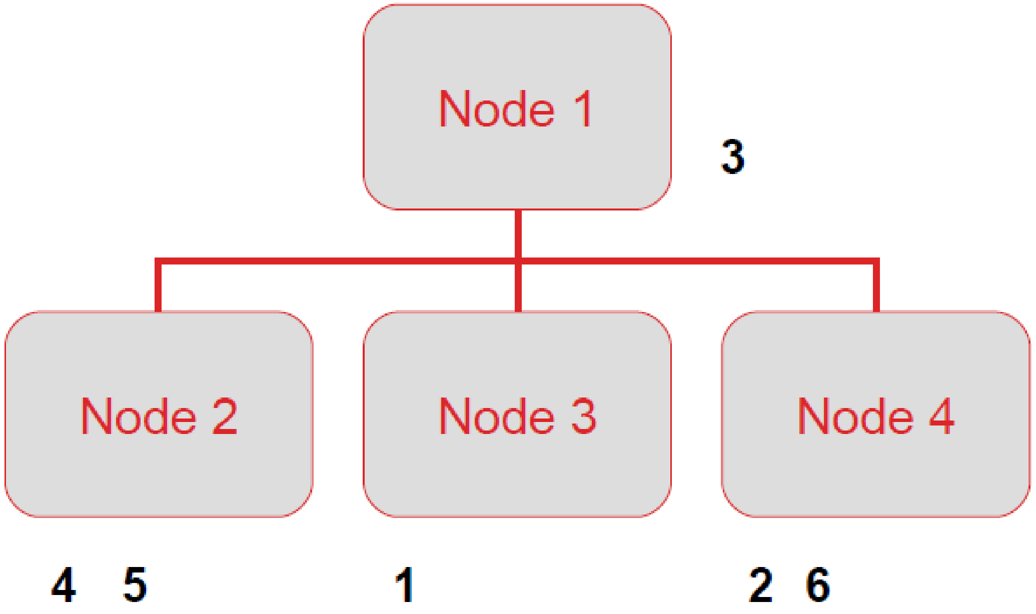
- Inserted data distributed for replicated tables
  - INSERT INTO region VALUES (1, ‘North America’, ‘comment1’);
  - …
  - INSERT INTO region VALUES (6, ‘Asia’, ‘comment6’);



Src: EnterpriseDB

Data distribution example

- Inserted data distributed for partitioned tables (suppose we change *region* to be a partitioned table)
  - INSERT INTO region VALUES (**1**, ‘North America’, ‘comment1’);
  - …
  - INSERT INTO region VALUES (**6**, ‘Asia’, ‘comment6’);



Src: EnterpriseDB

Inserting data

- INSERT INTO region VALUES (1, ‘North America’, ‘comment’);

- gs-loader
  - Uses COPY API
  - -b: basic checking like number of delimiters performed
  - -k: number of rows per “chunk” to try to load, percent reduction, smallest chunk size
  - Example:
    - **gs-loader.sh -d DEV -u admin -i /load/lineitem.tbl -t lineitem -b /load/dat/lineitem.dat -r # -k 100000,10,1 -y /load/dat**

Query processing

- Query Parsed
- Query Optimized
- Query Planned, Including Transformations
- Query Executed In Steps
  - Intelligently executes in parallel
  - First set of aggregates done in parallel at the nodes
  - Like, groups of intermediate results shipped to same target node
  - Second aggregation done in parallel
  - Coordinator streams in node results, combining on the fly and sending to client result set, performing a merge sort if ORDER BY present

Query Example 1

- SELECT COUNT(*) FROM ORDERS;

Query Example 1

- Step: 0

  Target: CREATE TABLE TMPTT1_1 (XCOL1 INT) WITHOUT OIDS

  Select: SELECT COUNT(*) AS XCOL1 FROM orders

- Step: 1

  Target:

  Select: SELECT SUM(XCOL1) AS EXPRESSION1 FROM TMPTT1_1

  Drop:

  TMPTT1_1

Query Example 2

- SELECT n_name, SUM(l_extendedprice)

```
FROM customer INNER JOIN orders on c_custkey = o_custkey
   INNER JOIN lineitem ON o_orderkey = l_orderkey
   INNER JOIN nation ON c_nationkey = n_nationkey
   INNER JOIN region ON n_regionkey = r_regionkey
 WHERE r_name = 'ASIA'
    AND c_mktsegment = 'BUILDING'
GROUP BY n_name
```

- Replicated: nation, region

- Partitioning columns: customer.c_custkey, orders.o_orderkey, lineitem.l_orderkey

Query Example 2

```
SELECT n_name, SUM(l_extendedprice)
FROM customer INNER JOIN orders on c_custkey = o_custkey
   INNER JOIN lineitem ON o_orderkey = l_orderkey
   INNER JOIN nation ON c_nationkey = n_nationkey
   INNER JOIN region ON n_regionkey = r_regionkey
WHERE r_name = 'ASIA'
   AND c_mktsegment = 'BUILDING'
GROUP BY n_name
```

Step: 0

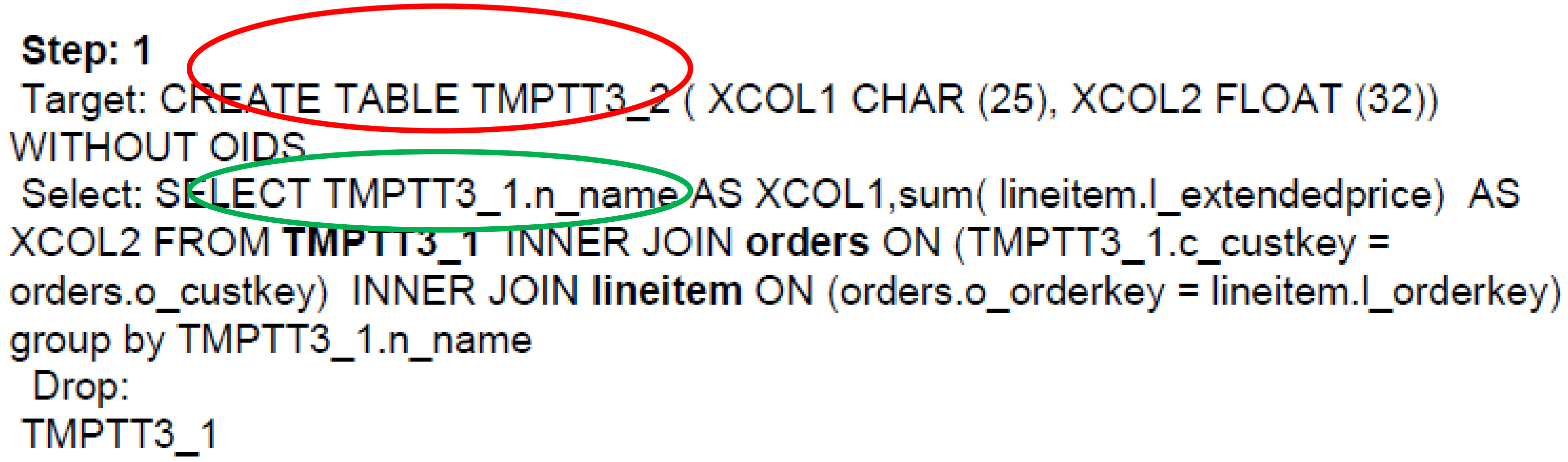
Target: CREATE TABLE TMPTT3_1 (n_name CHAR (25), c_custkey INT) WITHOUT OIDS

Select: SELECT nation.n_name AS n_name,customer.c_custkey AS c_custkey FROM **nation** INNER JOIN **region** ON (nation.n_regionkey = region.r_regionkey) INNER JOIN **customer** ON (customer.c_nationkey = nation.n_nationkey) WHERE (customer.c_mktsegment = 'BUILDING') AND (region.r_name = 'ASIA')

Query Example 2

```
SELECT n_name, SUM(l_extendedprice)
FROM customer INNER JOIN orders on c_custkey = o_custkey
   INNER JOIN lineitem ON o_orderkey = l_orderkey
   INNER JOIN nation ON c_nationkey = n_nationkey
   INNER JOIN region ON n_regionkey = r_regionkey
 WHERE r_name = 'ASIA'
    AND c_mktsegment = 'BUILDING'
GROUP BY n_name
```

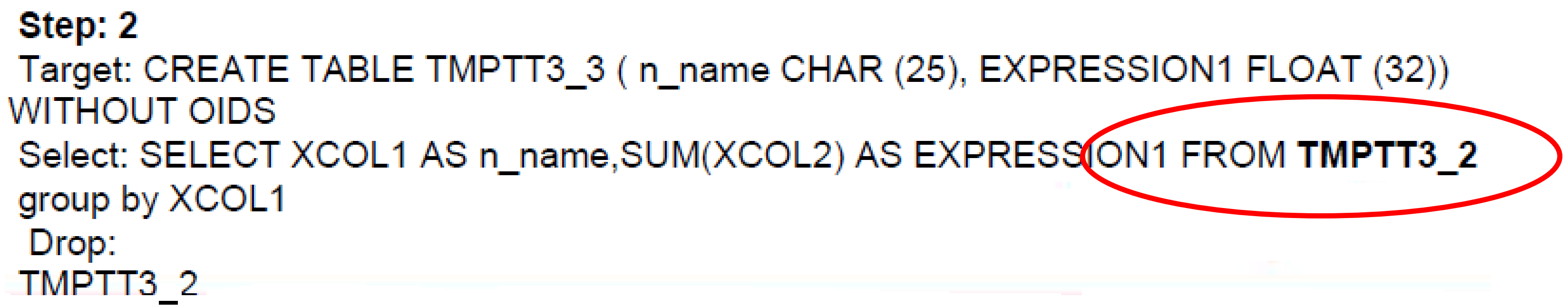
TMPTT3_1



```
 Step: 1
 Target: CREATE TABLE TMPTT3_2 ( XCOL1 CHAR (25), XCOL2 FLOAT (32))
WITHOUT OIDS
 Select: SELECT TMPTT3_1.n_name AS XCOL1,sum( lineitem.l_extendedprice)  AS
XCOL2 FROM TMPTT3_1  INNER JOIN orders ON (TMPTT3_1.c_custkey =
orders.o_custkey)  INNER JOIN lineitem ON (orders.o_orderkey = lineitem.l_orderkey)
group by TMPTT3_1.n_name
  Drop:
 TMPTT3_1
```

Query Example 2

```
SELECT n_name, SUM(l_extendedprice)
FROM customer INNER JOIN orders on c_custkey = o_custkey
   INNER JOIN lineitem ON o_orderkey = l_orderkey
   INNER JOIN nation ON c_nationkey = n_nationkey
   INNER JOIN region ON n_regionkey = r_regionkey
 WHERE r_name = 'ASIA'
    AND c_mktsegment = 'BUILDING'
GROUP BY n_name
```



```
Step: 2
Target: CREATE TABLE TMPTT3_3 ( n_name CHAR (25), EXPRESSION1 FLOAT (32))
WITHOUT OIDS
Select: SELECT XCOL1 AS n_name,SUM(XCOL2) AS EXPRESSION1 FROM TMPTT3_2
group by XCOL1
 Drop:
TMPTT3_2
```

Query Example 2

Step: 0

Target: CREATE TABLE TMPTT3_1 (n_name CHAR (25), c_custkey INT) WITHOUT OIDS

Select: SELECT nation.n_name AS n_name,customer.c_custkey AS c_custkey FROM **nation** INNER JOIN **region** ON (nation.n_regionkey = region.r_regionkey) INNER JOIN **customer** ON (customer.c_nationkey = nation.n_nationkey) WHERE (customer.c_mktsegment = 'BUILDING') AND (region.r_name = 'ASIA')

Step: 1

Target: CREATE TABLE TMPTT3_2 (XCOL1 CHAR (25), XCOL2 FLOAT (32)) WITHOUT OIDS

Select: SELECT TMPTT3_1.n_name AS XCOL1,sum(lineitem.l_extendedprice) AS XCOL2 FROM **TMPTT3_1** INNER JOIN **orders** ON (TMPTT3_1.c_custkey = orders.o_custkey) INNER JOIN **lineitem** ON (orders.o_orderkey = lineitem.l_orderkey) group by TMPTT3_1.n_name

Drop:

TMPTT3_1

Step: 2

Target: CREATE TABLE TMPTT3_3 (n_name CHAR (25), EXPRESSION1 FLOAT (32)) WITHOUT OIDS

Select: SELECT XCOL1 AS n_name,SUM(XCOL2) AS EXPRESSION1 FROM **TMPTT3_2** group by XCOL1

Drop:

TMPTT3_2

Limitations

- SQL Support
  - Uses its own parser and optimizer so:
    - No Window functions
    - No Stored Procedures
    - No Full-Text search
    - No Spatial support
- Transaction performance
  - Single row Insert, Update, or Delete are slow compared to a single PostgreSQL instance
  - The data must make an additional network trip to be committed

Limitations (contd.)

- Availability
  - No heartbeat or fail-over control in the coordinator
    - High Availability for each PostgreSQL node must be configured separately
    - Streaming replication can be ideal for this
  - Getting a consistent backup of the entire GridSQL database is difficult
    - Must ensure there are no transaction occurring
    - Backup each node separately

Limitations (contd.)

- Adding Nodes
  - Requires Downtime
    - Data must be manually reloaded to partition the data to the new node

References