Distributed Data Management

Christoph Lofi
Institut für Informationssysteme
Technische Universität Braunschweig
http://www.ifis.cs.tu-bs.de
3.0 Introduction

3.0 Query Processing
3.1 Basic Distributed Query Processing
3.2 Data Localization
3.3 Response Time Models
Architectures

• There are 3 major architectures for DDBMS
  – **Share-Everything Architecture**
    • Nodes share main memory
    • Suitably for tightly coupled high performance highly parallel DDBMS
    • Weaknesses wrt. scalability and reliability
  – **Shared-Disk Architecture**
    • Nodes have access to same secondary storage (usually **SAN**)
    • Strengths wrt. complex data and transactions
    • Common in enterprise level DDBMS
  – **Share-Nothing Architecture**
    • Node share nothing and only communicate over network
    • Common for web-age DDBMS and the cloud
    • Strength wrt. scalability and elasticity
• Data has to be distributed across nodes
• Main concepts:
  – **Fragmentation**: partition all data into smaller fragments / “chunks”
    • How to fragment? How big should fragments be? What should fragments contain?
  – **Allocation**: where should fragments be stored?
    • **Distribution** and replication
    • Where to put which fragment? Should fragments be replicated? If yes, how often and where?
In general, fragmentation and allocation are an optimization problem which are closely depended on the actual application:

- Focus on high availability?
- Focus on high degree of distribution?
- Focus on low communication costs and locality?
- Minimize or maximize geographic diversity?
- How complex is the data?
- Which queries are used how often?

Many possibilities and decision!
The task of DB query processing is to answer user queries
- e.g. “How many students are at TU BS in 2014?”
  - Answer: 17,200

However, some additional constraints must be satisfied
- Low response times
- High query throughput
- Efficient hardware usage
- …
- Relational Databases 2!
The generic workflow for centralized query processing involves multiple steps and components.
• **Query, Parser, and Naïve Query Plan**
  
  – Usually, the query is given by some **higher-degree declarative query language**
    
    • Most commonly, this is SQL
  
  – The query is translated by parser into some **internal representation**
    
    • Called naïve query plan
    
    • This plan is usually described by an **relational algebra operator tree**
• **Example:**
  
  – Database storing mythical creatures and heroes
    
    • Creature(cid, cname, type); Hero(hid, hname)
    
    • Fights(cid, hid, location, duration)
  
  – “Return the name of all creatures which fought at the Gates of Acheron”
    
    • **SELECT** cname **FROM** creature c, fights f
      WHERE c.cid=f.cid
      AND location=“Gates Of Acheron”
3.0 Query Processing

• Example (cont.)
  – Translate to relational algebra

  • $\pi_{cname} \sigma_{location='GoA'} \land creature.cid = fights.cid$ $\times$
    creature $\times$ fights

  • In contrast to the SQL statement, the algebra statement already contains the required basic evaluation operators
• Example (cont.)
  – Represent as operator tree

\[
\pi_{cname} \\
\sigma_{\text{location}='GoA' \land \text{creature.cid}=\text{fights.cid}} \times \\
\text{Creatures} \times \text{Fights}
\]
After the naïve query plan is found, the query rewriter performs simple transformations

- **Simple optimizations** which are always beneficial regardless of the system state (**Algebraic Query Rewriting**)
  - Mostly, **cleanup operations** are performed
  - Elimination of redundant predicates
  - Simplification of expressions
  - Normalization
  - Unnesting of subqueries and views
  - etc.

- **System state**
  - Size of tables
  - Existence or type of indexes
  - Speed of physical operations
  - etc.
The most effort in query preprocessing is spent on **query optimization**

- **Algebraic Optimization**
  - Find a better relational algebra operation tree
  - Heuristic query optimization
  - Cost-based query optimization
  - Statistical query optimization

- **Physical Optimization**
  - Find suitable *algorithms* for implementing the operations
• **Heuristic query optimization**
  – Use simple heuristics which usually lead to better performance
  – Basic Credo: **Not the optimal plan is needed, but the really crappy plans have to be avoided!**
  – Heuristics
    • **Break Selections**
      – Complex selection criteria should be broken into multiple parts
    • **Push Projection and Push Selection**
      – Cheap selections and projections should be performed as early as possible to reduce intermediate result size
    • **Force Joins**
      – In most cases, using a join is much cheaper than using a Cartesian product and a selection
• Example (cont.)
  – Perform algebraic optimization heuristics
    • Push selection & Projection
    • Force Join
• Most non-distributed RDBMS rely strongly on cost-based optimizations
  – Aim for better optimized plan which respect system and data characteristics
    • Especially, join order optimization is a challenging problem
  – Idea
    • Establish a cost model for various operations
    • Enumerate all query plans
    • Pick best query plan
  – Usually, **dynamic programming** techniques are used to keep computational effort manageable
• **Algebraic optimization** results in an optimized query plan which is still represented by relational algebra

  – **How is this plan finally executed?**
    - > Physical Optimization
  
  – There are multiple algorithms for implementing a given relational algebra operation
    - **Join**: Block-Nested-Loop join? Hash-join? Merge-Join? etc.
    - **Select**: Scan? Index-lookup? Ad-hoc index generation & lookup? etc.
    - ...

3.0 Query Processing
Physical optimization translates the query plan into an execution plan

- Which algorithms will be used on which data and indexes?
  - Physical Relational Algebra
    - For each standard algebra operation, several physical implementations are offered
  - Should pipelining (iterator model) be used? How?
- Physical and algebraic optimization are often tightly interleaved
  - Most physical optimization also relies on cost-models
  - Idea: perform cost-based optimization algorithm “in one sweep” for algebraic, join order, and physical optimization
• Example (cont.)

Filter all but cname

Index Nested Loop Join

Primary Index Of Creatures

Table Creatures

Index Key Lookup: fights.location="GoA"
In general, centralized non-distributed query processing is a well understood problem

- Current problems
  - Designing effective and efficient cost models
  - Eliciting meaningful statistics for use in cost models
  - Allow for database self-tuning
• Distributed query processing (DQP) shares many properties of centralized query processing
  – Basically, the same problem…
  – But: objectives and constraints are different!

• Objectives for centralized query processing
  – Minimize number of disk accesses!
  – Minimize computational time
Objectives for distributed query processing are usually less clear...

- Minimize resource consumption?
- Minimize response time?
- Maximize throughput?

Additionally, costs are more difficult to predict

- Hard to elicit meaningful statistics on network and remote node properties
  - Also, high variance in costs
• Additional **cost factors, constraints, and problems** must be respected
  – Extension of physical relational algebra
    • Sending and receiving data
  – Data localization problems
    • Which node holds the required data?
  – Deal with replication and caching
  – Network models
  – Response-time models
  – Data and structural heterogeneity
    • Think federated database…
3.1 Basic DQP

- Often, **static enumeration optimizations** do are difficult in distributed setting
  - More difficult than non-distributed optimization
    - More and conflicting **optimization goals**
    - Unpredictability of costs
    - More costs factors and constraints
    - Quality-of-Service agreements (QoS)
  - Thus, most successful queries optimization techniques are **adaptive**
    - Query is optimized **on-the-fly** using current, directly measured information of the system’s behavior and workload
  - Don’t target for the **best plan**, but for a **good plan**
Example distributed query processing:

- “Find the creatures involved in fights being decided in exactly one minute”
  
  \[ \pi_{\text{cname}} \sigma_{\text{duration}=1\text{min}} \text{creature} \bowtie \text{fights} \]

Problem:

- Relations are fragmented and distributed across five nodes
- The creature relation uses primary horizontal partitioning by creature type
  - One fragment of the relation is located at node 1, the other on node 2; no replication
- The fights relation uses derived horizontal partitioning
  - One fragment on node 3, one on node 4; no replication
- Query originates from node 5
3.1 Basic DQP

Query

N1

Creatures₁

N2

N3

N4

N5

Fights₂

Creatures₂

Fights₁
• **Cost model and relation statistics**
  – Accessing a tuple (\(tupacc\)) costs 1 unit
  – Transferring a tuple (\(tuptrans\)) costs 10 units
  – There are 400 creatures and 1000 fights
    • 20 fights are one minute
    • All tuples are uniformly distributed
      – i.e. node 3 and 4 contain 10 short-fight-tuples each
  – There are local indexes on attribute “duration”
    • Direct tuple access possible on a local sites, no scanning
  – All nodes may directly communicate with each other
3.1 Basic DQP

\[ \pi_{\text{cname}} \sigma_{\text{duration}=1\text{min}} \text{creature} \bowtie \text{fights} \]
• Two simple distributed query plans
  – Version A: Transfer all data to node 5

\[
\pi_{cname} (\text{Creature}_1 \cup \text{Creature}_2) \bowtie (\sigma_{duration=1\,\text{min}} \text{Fights}_1 \cup \text{Fights}_2)
\]
3.1 Basic DQP

– Version B: ship intermediate results

\[ \pi_{\text{cname}}(\text{Creature}_1^* \cup \text{Creature}_2^*) \]

Node 5

\[ \sigma_{\text{duration}=1\text{min}} Fights_1^* \rightarrow \text{Receive } Fights_1^* \]

Node 1

\[ \text{Send Creature}_1^* \rightarrow \text{Receive } Fights_1^* \]

Node 2

\[ \text{Send Creature}_2^* \rightarrow \text{Receive } Fights_2^* \]

Node 3

\[ \text{Send } Fights_1^* \rightarrow \text{Receive } Fights_1^* \]

Node 4

\[ \text{Send } Fights_2^* \rightarrow \text{Receive } Fights_2^* \]

\[ \sigma_{\text{duration}=1\text{min}} Fights_2^* \]
3.1 Basic DQP

- Costs A: 23,000 Units

\[
\pi_{cname} (Creature_1 \cup Creature_2) \bowtie (\sigma_{duration=1min} Fights_1 \cup Fights_2)
\]

Node 5

Receive Creature_1

Receive Creature_1

Receive Fights_1

Receive Fights_2

Send Creature_1

Send Creature_2

Send Fights_1

Send Fights_2

400*20*tupacc = 8,000 (BNL join)

1,000*tupacc = 1,000 (selection w/o index)

200*tuptrans = 2,000

200*tuptrans = 2,000

500*tuptrans = 5,000

500*tuptrans = 5,000

\text{tupacc}=1;\ \text{tuptrans}=10
3.1 Basic DQP

- Cost B: 460 Units

\[ \pi_{cname} \left( \text{Creature}_1^* \cup \text{Creature}_2^* \right) \]

\[ \sigma_{duration=1\text{min}Fights_1} \]

Node 3

\[ \text{Send Fights}_1^* \]

Node 1

\[ \text{Send Creature}_1^* \]

Node 5

\[ \pi_{cname} \left( \text{Creature}_1^* \cup \text{Creature}_2^* \right) \]

Node 2

\[ \text{Send Creature}_2^* \]

Node 4

\[ \text{Send Fights}_2^* \]

Node 1

\[ \text{Receive Fights}_1^* \]

Node 2

\[ \text{Receive Fights}_2^* \]

\[ \text{Node 5} \]

\[ 10^{*}\text{tuptrans} = 100 \]

\[ 20^{*}\text{tupacc} = 20 \]

\[ 10^{*}\text{tuptrans} = 100 \]

\[ 20^{*}\text{tupacc} = 20 \]

\[ 10^{*}\text{tuptrans} = 100 \]

\[ 10^{*}\text{tupacc} = 10 \]

\[ 10^{*}\text{tuptrans} = 100 \]

\[ 20^{*}\text{tupacc} = 10 \]

\[ \text{tupacc}=1;\ \text{tuptrans}=10 \]
3.1 Basic DQP

• For performing any query optimization **meta data** is necessary
  – Meta data is stored in the **catalog**
  – The catalog of a DDBMS especially has to store information on data distribution

• Typical **catalog** contents
  – **Database schema**
    • Definitions of tables, views, UDFs & UDTs, constraints, keys, etc.
3.1 Basic DQP

– **Partitioning schema**
  - Information on how the schema is *partitioned* and how tables can be *reconstructed*

– **Allocation schema**
  - Information on where which fragment can be found
  - Thus, also contains information on *replication* of fragments

– **Network information**
  - Information on node connections
  - Connection information (network model)

– **Additional physical information**
  - Information on indexes, data statistics (basic statistics, histograms, etc.), hardware resources (processing & storage), etc.
Central problem in DDBMS: Where and how should the catalog be stored?

- Simple solution: store at a central node
- For slow networks, it beneficial to replicate the catalog across many / all nodes
  - Assumption: catalog is small and does not change often
- Also, caching is a viable solution
  - Replicate only needed parts of a central catalog, anticipate potential inconsistencies
- In rare cases, the catalog may grow very large and may change often
  - Catalog has to be fragmented and allocated
  - New meta problem: where to find which catalog fragment?
• What should be optimized when and where?
  – We assume that most applications use **canned queries**
    • i.e. prepared and parameterized SQL statements

• **Full compile time-optimization**
  – Similar to centralized DBs, the full query execution plan is computed at compile time
  – **Pro:**
    • Queries can be directly executed
  – **Con:**
    • Complex to model
    • Many information unknown or too expensive to gather (collect statistics on all nodes?)
    • Statistics outdated
      – Especially, machine load and network properties are very volatile
3.1 Basic DQP

- **Fully dynamic optimization**
  - Every query is optimized individually at *runtime*
  - Heavily relies on heuristics, learning algorithms, and luck

- **Pro**
  - Might produce very good plans
  - Uses current network state
  - Also usable for ad-hoc queries

- **Con**
  - Might be very unpredictable
  - Complex algorithms and heuristics
• **Semi-dynamic and hierarchical approaches**

  – Most DDBMS optimizers use semi-dynamic or hierarchical optimization techniques (or both)

  – **Semi-dynamic**
    
    • **Pre-optimize** the query
    
    • During query execution, **test** if execution follows the plan
      
      – e.g. if tuples/fragments are delivered in time, if network has predicted properties, etc.
    
    • If execution shows severe plan **deviations**, compute a new query plan for all missing parts
3.1 Basic DQP

• Hierarchical Approaches
  – Plans are created in multiple stages
  – Global-Local Plans
    • Global query optimizer creates a global query plan
      – i.e. focus on data transfer: which intermediate results are to be computed by which node, how should intermediate results be shipped, etc.
    • Local query optimizers create local query plans
      – Decide on query plan layout, algorithms, indexes, etc. to deliver the requested intermediate result
Two-Step-Plans

- During compile time, only stable parts of the plan are computed
  - Join order, join methods, access paths, etc.
- During query execution, all missing plan elements are added
  - Node selection, transfer policies, …
- Both steps can be performed using traditional query optimization techniques
  - Plan enumeration with dynamic programming
  - Complexity is manageable as each optimization problem is much easier than a full optimization
  - During runtime optimization, fresh statistics are available
3.2 Data Localization

- The first problem in distributed query processing is **data localization**
  
  - **Problem:** *query transparency* is needed
    - User queries the global schema
    - However, the relations of global schema are fragmented and distributed

  - **Assumption**
    - Fragmentation is given by **partitioning rules**
      - *Selection* predicates for horizontal partitioning
      - Attribute *projection* for vertical partitioning
    - Each fragment is allocated only at one node
      - *No replication*
    - Fragmentation rules and location of the fragments is stored in catalog
3.2 Data Localization

- **Base Idea:**
  - **Query Rewriter** is modified such that each query to **global schema** is replaced by a query on the **distributed schema**
    - i.e. each reference to a global relation is replaced by a **localization program** which **reconstructs** the table
  - If the localization program reconstructs the **full relation**, this is called a **generic query**
  - Often, the full relation is not necessary and by inspecting the query, simplifications can be performed
    - **Reduction techniques** for the localization program
### 3.2 Data Localization

- **Example:**
  
  - Relation $\textit{Creature} = (\text{cid}, \text{cname}, \text{type})$
  
  - Primary partitioning by id
    
    - $\textit{Cr}t_1 = \sigma_{\text{cid} \leq 100} \textit{Creature}$
    - $\textit{Cr}t_2 = \sigma_{100 < \text{cid} < 200} \textit{Creature}$
    - $\textit{Cr}t_3 = \sigma_{\text{cid} \geq 200} \textit{Creature}$

  - Allocate each fragment to its own node
    
    - $\textit{Cr}t_1$ to node 1, ...

  - A generic localization program for $\textit{Creature}$ is given by
    
    - $\textit{Creature} = \textit{Cr}t_1 \cup \textit{Cr}t_2 \cup \textit{Cr}t_3$
• Global queries can now easily be transformed to generic queries by replacing table references
  – SELECT * FROM creatures WHERE cid = 9
  • Send and receive operations are implicitly assumed
3.2 Data Localization

• Often, when using generic queries, **unnecessary fragments** are transferred and accessed
  – We know the **partitioning rules**, so it is clear that the requested tuple is in fragment $Crt_1$
  – Use this knowledge to **reduce** the query
In general, the **reduction rule for primary horizontal partitioning** can be stated as

- Given fragments of $R$ as $F_R = \{R_1, \ldots, R_n\}$ with $R_j = \sigma_{P_j}(R)$

**Reduction Rule 1:**

- All fragments $R_j$ for which $\sigma_{p_s}(R_j) = \emptyset$ can be omitted from localization program
  - $p_s$ is the query selection predicate
  - e.g. in previous example, $cid = 9$ contradicts $100 < cid < 200$

- $\sigma_{p_s}(R_j) = \emptyset \iff \forall x \in R: \neg(p_s(x) \land p_j(x))$

- “The selection with the query predicate $p_s$ on the fragment $R_j$ will be empty if $p_s$ contradicts the partitioning predicate $p_j$ of $R_j”$
  - i.e. $p_s$ and $p_j$ are never true at the same time for all tuples in $R_j$
• **Join Reductions**
  
  – Similar reductions can be performed with queries involving a `join` and relations partitioned along the `join` attributes
  
  – Base Idea: Larger joins are replaced by **multiple partial joins** of fragments

  \[
  (R_1 \cup R_2) \bowtie S \equiv (R_1 \bowtie S) \cup (R_2 \bowtie S)
  \]

  • Which might or might not be a good idea depending on the data or system
  
  • **Reduction**: Eliminate all those unioned fragments from evaluation which will return an empty result
3.2 Data Localization

- i.e.: join according to the join graph
  - Join graph usually not known (full join graph assumed)
    - Discovering the non-empty partial joins will construct join graph
3.2 Data Localization

• We hope for
  – …many partial joins which will definitely produce empty results and may be omitted
    • This is not true if partitioning conditions are suboptimal
  – …many joins on small relations have lower resource costs than one large join
    • Also only true if “sensible” partitioning conditions used
    • Not always true, depends on used join algorithm and data distributions; still a good heuristic
  – …smaller joins may be executed in parallel
    • Again, this is also not always a good thing
    • May potentially decrease response time…
      – Response time cost models!
    • …but may also increase communication costs
3.2 Data Localization

• Example:

  – *Creature* = (*cid*, *cname*, *type*)
    - *Crt*₁ = *σ*₁₀₀≤*cid* *Creature*
    - *Crt*₂ = *σ*₁₀₀<*cid*<₂₀₀ *Creature*
    - *Crt*₃ = *σ*₂₀₀≥*cid* *Creature*

  – *Fights* = (*cid*, *hid*, *location*, *duration*)
    - *Fg*₁ = *σ*₂₀₀≤*cid* *Fights*
    - *Fg*₂ = *σ*₂₀₀>*cid* *Fights*

  – SELECT * FROM * creature *c*, *fight* *f*  
    WHERE *c.cid* = *f.cid*
3.2 Data Localization

- By splitting the joins, six fragment joins are created
- 3 of those fragment joins are empty
  - e.g. $\text{Crt}_1 \bowtie \text{Fg}_2 = \emptyset$
  - $\text{Crt}_1$ contains tuples with cid $\leq 100$
  - $\text{Fg}_2$ contains tuples with cid $> 200$
• Formally
  
  – Join Fragmentation
    • \((R_1 \cup R_2) \bowtie S \equiv (R_1 \bowtie S) \cup (R_2 \bowtie S)\)
  
  – **Reduction Rule 2:**
    \[ R_i \bowtie R_j = \emptyset \iff \forall x \in R_i, y \in R_j: \neg(p_i(x) \land p_j(y)) \]
    • “The join of the fragments \(R_j\) and \(R_i\) will be empty if their respective partition predicates (on the join attribute) contradict.”
      
      – i.e. there is no tuple combination \(x\) and \(y\) such that both partitioning predicates are fulfilled at the same time
  
  • **Empty join fragments may be reduced**
3.2 Data Localization

- Obviously, the easiest join reduction case follows from **derived horizontal fragmentation**
  - For each fragment of the first relation, there is **exactly one matching fragment** of the second relation
    - The **reduced query** will always be more **beneficial** than the generic query due to small number of fragment joins
  - Derived horizontal fragmentation is especially effective to represent **one-to-many** relationships
    - Many-to-many relationships are only possible if tuples are replicated
      - No fragment disjointness!
3.2 Data Localization

- **Example:**
  
  - *Creature* = *(cid, cname, type)*
    - *Crt*$_1$ = $\sigma_{cid \leq 100}$ *Creature*
    - ... 
  
  - *Fights* = *(cid, hid, location, duration)*
    - *Fg*$_1$ = *Fights* $\bowtie$ *Creature*
    - ... 
  
  - **SELECT** * * **FROM** *creature c, fight f*  
    **WHERE** *c.cid = f.cid*
• Reduction for **vertical fragmentation** is very similar

  – **Localization program** for $R$ is usually of the form
    
    • $R = R_1 \bowtie R_2$

  – When reducing generic vertically fragmented queries, avoid joining in fragments containing useless attributes

  – **Example:**
    
    • $\text{Creature} = (\text{cid}, \text{cname}, \text{type})$ is fragmented to $\text{Creature}_1 = (\text{cid}, \text{cname})$ and $\text{Creature}_2 = (\text{cid}, \text{type})$
    
    • For the query **SELECT** $\text{cname}$ **FROM** $\text{creature}$ **WHERE** $\text{cid}=9$, no access to $\text{Creature}_2$ is necessary
3.2 Data Localization

• Reducing Queries w. Hybrid Fragmentation

  – Localization program for \( R \) combines joins and unions

    • e.g. \( R = (R_1 \cup R_2) \bowtie R_3 \)

  – General guidelines are

    • Remove empty relations generated by contradicting selections on horizontal fragments
      – Relations containing useless tuples
    • Remove useless relations generated by vertical fragments
      – Relations containing unused attributes
    • Break and distribute joins, eliminate empty fragment joins
      – Fragment joins with guaranteed empty results
• Previously, we computed **reduced queries** from global queries

• However, where should the query be executed?
  – Assumption: only two nodes involved
    • i.e. **client-server setting**
    • Server stores data, query originates on client

  – **Query shipping**
    • Common approach for centralized DBMS
    • Send query to the server node
    • Server computes the query result and ships result back
### 3.2 Data Localization

#### Data Shipping
- Query remains at the client
- Server ships all required data to the client
- Client computes result
3.2 Data Localization

• **Hybrid Shipping**
  – Partially send query to server
  – Execute some query parts at the server, send intermediate results to client
  – Execute remaining query at the client
3.2 Data Localization

• Of course, these simple models can be extended to **multiple nodes**
  – Query optimizer has to decide which parts of the query have to be shipped to which node
    • Cost model!
  – In heavily replicated scenarios, clever hybrid shipping can effectively be used for **load balancing**
    • Move expensive computations to lightly loaded nodes
    • Avoid expensive communications
3.3 Response Time Models

• “Classic” DB cost models focus on total resource consumption of a query
  – Leads to good results for heavy computational load and slow network connections
    • If query saves resources, many queries can be executed in parallel on different machines
  – However, queries can also be optimized for short response times
    • “Waste” some resources to get query results earlier
    • Take advantage of lightly loaded machines and fast connections
    • Utilize intra-query parallelism
      – Parallelize one query instead of multiple queries
3.3 Response Time Models

- **Response time models** are needed!
  - “When does the *first result tuple* arrive?”
  - “When have all *tuples* arrived?”

- **Example**
  - Assume relations or fragments A, B, C, and D
  - All relations/fragments are available on all nodes
    - Full replication
  - Compute \((A \bowtie B) \bowtie (C \bowtie D)\)

- **Assumptions**
  - Each join costs 20 time units (TU)
  - Transferring an intermediate result costs 10 TU
  - Accessing relations is free
  - Each node has one computation thread
3.3 Response Time Models

• Two plans:
  – Plan 1: Execute all operations on one node
    • Total costs: 60
  – Plan 2: Join on different nodes, ship results
    • Total costs: 80
• With respect to total costs, plan 1 is better

• Example (cont.)
  – But: Plan 2 is better wrt. to \textit{response time} as operations can be carried out in parallel
3.3 Response Time Models

• **Response Time**
  - Two types of response times
    • **First Tuple & Full Result** Response Time

• **Computing response times**
  - Sequential execution parts
    • Full response time is sum of all computation times of all used operations
  - Multiple parallel threads
    • Maximal costs of all parallel sequences
• Considerations:
  – How much speedup is possible due to parallelism?
    • Or: “Does kill-it-with-iron” work for parallel problems?
  – Performance speed-up of algorithms is limited by Amdahl’s Law
    • Gene Amdahl, 1968
    • Algorithms are composed of parallel and sequential parts
    • Sequential code fragments severely limit potential speedup of parallelism!
– Possible maximal speed-up:
  
  \[ \text{maxspeedup} \leq \frac{p}{1 + s(p-1)} \]

  • \( p \) is number of parallel threads
  • \( s \) is percentage of single-threaded code

– e.g. if 10% of an algorithm is sequential, the maximum speed up regardless of parallelism is 10x

– For maximal efficient parallel systems, all sequential bottlenecks have to be identified and eliminated!
3.3 Response Time Models

Amdahl’s Law

- Parallel Portion
  - 50%
  - 75%
  - 90%
  - 95%

Number of Processors

Speedup

1 2 4 8 16 32 64 128 256 512 1024 2048 4096 8192 16384 32768 65536
**3.3 Response Time Models**

- **Good First Tuple Response** benefits from queries executed in a pipelined fashion

  - **Not pipelined:**
    - Each operation is fully completed and an *intermediate result* is created
    - Next operation reads intermediate result and is then fully completed
      - Reading and writing of intermediate results costs resources!

  - **Pipelined**
    - Operations *do not create intermediate* results
    - Each finished tuple is fed directly into the next operation
    - Tuples “flow” through the operations
• Usually, the tuple flow is controlled by iterator interfaces implemented by each operation
  – “Next tuple” command
  – If execution speed of operations in the pipeline differ, tuples are either cached or the pipeline blocks

• Some operations are more suitable than others for pipelining
  – **Good**: scan, select, project, union, …
  – **Tricky**: join, intersect, …
  – **Very Hard**: sort
### 3.3 Response Time Models

- **Simple pipeline example:**
  - **Tablescan, Selection, Projection**
    - 1000 tuples are scanned, selectivity is 0.1
  - **Costs:**
    - Accessing one tuple during tablescan: 2 TU (time unit)
    - Selecting (testing) one tuple: 1 TU
    - Projecting one tuple: 1 TU

#### Non-Pipelined

- **IR1** → **Selection** → **Projection** → **Final**

#### Pipelined

<table>
<thead>
<tr>
<th>time</th>
<th>event</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>First tuple finished tablescan</td>
</tr>
<tr>
<td>3</td>
<td>First tuple finished selection (if selected...)</td>
</tr>
<tr>
<td>4</td>
<td>First tuple in Final</td>
</tr>
<tr>
<td>3098</td>
<td>Last tuple finished tablescan</td>
</tr>
<tr>
<td>3099</td>
<td>Last tuple finished selection</td>
</tr>
<tr>
<td>3100</td>
<td>All tuples in Final</td>
</tr>
</tbody>
</table>
• Consider following example:
  – Joining two table subsets
    • Non-pipelined BNL join
    • Both pipelines work in parallel
  – Costs:
    • 1,000 tuples are scanned in each pipeline, selectivity 0.1
    • Joining \(100 \bowtie 100\) tuples: 10,000 TU (1 TU per tuple combination)
  – **Response time** (non-pipelined BNL)
    • The first tuple can arrive at the end of any pipeline after 4 TU
      – Stored in intermediate result
    • All tuples have arrived at the end of the pipelines after 3,100 TU
    • Final result will be available after 13,100 TU
      – No benefit from pipelining wrt. response time
      – First tuple arrives at \(3100 \ll t \leq 13100\)
• The suboptimal result of the previous example is due to the **unpipelined join**
  – Most traditional join algorithms are unsuitable for pipelining
    • Pipelining is not usually necessary feature in a strict *single thread environment*
      – Join is fed by two input pipelines
      – Only one pipeline can be executed at a time
      – Thus, at least one intermediate result has to be created
      – Join may be performed *single / semi-pipelined*
    • In parallel / distributed DBs, fully pipelined joins are beneficial
3.3 Response Time Models

• **Single-Pipelined-Hash-Join**
  – One of the “classic” join algorithms
  – **Base idea** $A \bowtie B$
    • One input relation is read from an **intermediate result** ($B$), the other is **pipelined** though the join operation ($A$)
    • All tuples of $B$ are stored in a hash table
      – **Hash function** is used on the **join attribute**
      – i.e. all tuples showing the same value for the join attribute are in one bucket
        » Careful: **hash collisions**! Tuple with different joint attribute value might end up in the same bucket!
    • Every incoming tuple $a$ (via pipeline) of $A$ is **hashed** by join attributed
    • **Compare** $a$ to each tuple in the respective $B$ bucket
      – Return those tuples which show matching join attributes
3.3 Response Time Models

- **Double-Pipelined-Hash-Join**
  - Dynamically build a hashtable for A and B each
    - **Memory intensive!**
  - Process tuples on arrival
    - Cache tuples if necessary
    - Balance between A and B tuples for better performance
    - Rely on statistics for a good A:B ratio
  - If a new A tuple a arrives
    - Insert a into the A-table
    - Check in the B table if there are join partners for a
    - If yes, return all matching AB tuples
  - If a new B tuple arrives, process it analogously
3.3 Response Time Models

- \( \text{B}(31, B2) \) arrives
- Insert into \( \text{B} \) Hash
- Find matching \( \text{A} \) tuples
  - Find \( \text{A}3 \)
  - Assume that \( \text{A}3 \) matches \( \text{B}2 \)...
- Put \( \text{AB}(\text{A}3, \text{B}2) \) into output feed
• In pipelines, tuples just “flow” through the operations
  – No problem with that in one processing unit…
  – But how do tuples flow to other nodes?
• Sending each tuple individually may be very ineffective
  – Communication costs:
    • Setting up transfer & opening communication channel
    • Composing message
    • Transmitting message: header information & payload
      – Most protocols impose a minimum message size & larger headers
      – Tuplesize \ll \text{Minimal Message Size}
    • Receiving & decoding message
    • Closing channel
3.3 Response Time Models

• Idea: Minimize Communication Overhead by **Tuple Blocking**
  – Do not send single tuples, but larger blocks containing multiple tuples
    • “Burst-Transmission”
    • Pipeline-Iterators have to be able to cache packets
    • Block size should be at least the packet size of the underlying network protocol
      – Often, larger packets are more beneficial
      – ….more cost factors for the model
• **Additional** constraints and **cost factors** compared to “classic” query optimization
  – Network **costs**, network **model**, **shipping policies**
  – **Fragmentation & allocation** schemes
  – Different optimization goals
    • **Response time vs. resource consumption**

• **Basic techniques** try to **prune** unnecessary accesses
  – Generic query reductions
• This lecture only covers very basic techniques
  – In general, distributed query processing is a very complex problem
  – Many and new optimization algorithms are researched
    • Adaptive and learning optimization
    • Eddies for dynamic join processing
    • Fully dynamic optimization
    • …

• Recommended literature
• Distributed Transaction Management
  – Transaction Synchronization
  – Distributed Two-Phase Commits
  – Byzantine Agreements