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
    - Suitable 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

Fragmentation

- 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?

Fragmentation

- In general, fragmentation and allocation are 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!

3.0 Query Processing

- The task of DB query processing is to answer user queries
  - e.g. “How many students are at TU BS in 2009?”
    - Answer: 14,100
- 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.

**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"

After the naïve query plan is found, the query rewriter performs simple transformations.

- **Example (cont.)**
  - Represent as operator tree

- **Example (cont.)**
  - Translate to relational algebra
    - $\pi_{\text{cname}} (\sigma_{\text{location} = \text{'Gates Of Acheron'}} \text{creature} \times \text{fights})$
    - In contrast to the SQL statement, the algebra statement already contains the required basic evaluation operators.

- **Example (cont.)**
  - System state
    - Size of tables
    - Existence or type of indexes
    - Speed of physical operations
    - etc.
3.0 Query Processing

- 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

3.0 Query Processing

- 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

3.0 Query Processing

- 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

3.0 Query Processing

- Example (cont.)
  - Perform algebraic optimization heuristics
    - Push selection & Projection
    - Force Join

3.0 Query Processing

- 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
    - 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
3.0 Query Processing

• Example (cont.)

3.1 Basic DQP

• 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

3.1 Basic DQP

• 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

3.1 Basic DQP

• 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 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”

\[ \pi_{\text{name}} (\sigma_{\text{duration}=1\text{min}} \text{Creature} \bowtie \text{fights}) \]

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

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

Two simple distributed query plans

- Version A: Transfer all data to node 5

\[ \pi_{\text{name}} (\text{Creature}_{1} \cup \text{Creature}_{2}) \bowtie (\sigma_{\text{duration}=1\text{min}} \text{Fights}_{1} \cup \text{Fights}_{2}) \]

- Version B: ship intermediate results

\[ \pi_{\text{name}} (\text{Creature}_{1} \cup \text{Creature}_{2}) \]

Node 5

- Receive `Creature_{1}`
- Receive `Creature_{2}`
- Receive `Fights_{1}`
- Receive `Fights_{2}`

Node 1

- Send `Creature_{1}`
- Send `Creature_{2}`
- Send `Fights_{1}`
- Send `Fights_{2}`

Node 2

- Send `Creature_{1}`
- Send `Creature_{2}`
- Send `Fights_{1}`
- Send `Fights_{2}`

Node 3

- Send `Fights_{1}`
- Receive `Creature_{1}`
- Receive `Fights_{1}`

Node 4

- Send `Fights_{2}`
- Receive `Creature_{2}`
- Receive `Fights_{2}`
### 3.1 Basic DQP

**Detour**

- Costs A: 23,000 Units
  - Partitioning schema
    - Prepared and parameterized SQL statements
  - Full compile time
    - Information on indexes, data statistics (basic statistics, histograms, etc.)
  - Con:
    - Prepared and parameterized SQL statements
    - In rare cases, the catalog may grow very large and may change often
    - 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

**Defeat**

- Cost B: 460 Units
  - Receive
    - Definitions of tables, views, UDFs & UDTs, constraints, keys, etc.
  - Send
    - Information on node connections
    - Information on network model
    - Information on indexes, data statistics (basic statistics, histograms, etc.), hardware resources (processing & storage), etc.
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

3.1 Basic DQP

• 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

3.1 Basic DQP

• 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 Creature = (cid, cname, type)
  – Primary partitioning by id
    • Crt1 = σ cid=1σ creature
    • Crt2 = σ cid=2σ creature
    • Crt3 = σ cid=3σ creature
  – Allocate each fragment to its own node
    • Crt1 to node 1, ...
  – A generic localization program for Creature is given by
    • Creature = Crt1 U Crt2 U Crt3

• 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 Crt1
  – Use this knowledge to reduce the query

• 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
    • (R1 U R2) S \equiv (R1 S) U (R2 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

• 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

• In general, the reduction rule for primary horizontal partitioning can be stated as
  – Given fragments of R as FR = {R1, ..., Rn} with Rj = σpj(R)
  – Reduction Rule 1:
    • All fragments Rj for which σpj(Rj) = ∅ can be omitted from localization program
      – pj is the query selection predicate
      – e.g. in previous example, cid = 9 contradicts 100 < cid < 200
    • σpj(Rj) = ∅ ⇔ ∀ x ∈ Rj: ¬(pj(x) ∧ pj(x))
    • “The selection with the query predicate pj on the fragment Rj will be empty if pj contradicts the partitioning predicate p_j of Rj”
      – i.e. pj and p_j are never true at the same time for all tuples in Rj

• 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 model!
    - ... but may also increase communication costs

- 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 disjunctivity!

3.2 Data Localization

- Example:
  - \( \text{Creature} = (\text{cid}, \text{cname}, \text{type}) \)
    - \( \text{Cri}_1 = \sigma_{\text{cid}=100}\text{Creature} \)
    - \( \text{Cri}_2 = \sigma_{\text{cid}<200}\text{Creature} \)
    - \( \text{Cri}_3 = \sigma_{\text{cid}=200}\text{Creature} \)
  - \( \text{Fights} = (\text{cid}, \text{hid}, \text{location}, \text{duration}) \)
    - \( \text{Fg}_1 = \sigma_{\text{cid}=200}\text{Fights} \)
    - \( \text{Fg}_2 = \sigma_{\text{cid}=100}\text{Fights} \)
    - \( \text{SELECT} * \text{FROM} \text{creature c, fight f WHERE} \text{c.cid=f.cid} \)

- Formally
  - \( \text{Join Fragmentation} \)
    - \( (R_1 \cup R_2) \bowtie S \equiv (R_1 \bowtie S) \cup (R_2 \bowtie S) \)
  - \( \text{Reduction Rule 2:} \)
    - \( R_1 \bowtie R_2 = \emptyset \Rightarrow \forall x \in R_1, y \in R_2; - (p_1(x) \land p_2(y)) \)
    - “The join of the fragments \( R_1 \) and \( R_2 \) 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

- Example:
  - \( \text{Creature} = (\text{cid}, \text{cname}, \text{type}) \)
    - \( \text{Cri}_1 = \sigma_{\text{cid}=100}\text{Creature} \)
    - ...  
  - \( \text{Fights} = (\text{cid}, \text{hid}, \text{location}, \text{duration}) \)
    - \( \text{Fg}_1 = \text{Fights} \bowtie\text{Creature} \)
    - ...  
    - \( \text{SELECT} * \text{FROM} \text{creature c, fight f WHERE} \text{c.cid=f.cid} \)
3.2 Data Localization

- **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:**
    - $Creatue = (cid, name, type)$ is fragmented to $Creatue_1 = (cid, name)$ and $Creatue_2 = (cid, type)$
    - For the query $\text{SELECT name FROM creature WHERE cid}=9$, no access to $Creatue_2$ is necessary

- **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

- 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

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

- **Data Localization**
  - 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

- **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

- **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
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

- 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

- 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 response time as operations can be carried out in parallel

- 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!
3.3 Response Time Models

- 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

- **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 the 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

3.3 Response Time Models

- 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:
  - **Non-pipelined** BNL join
    - Accessing one tuple during tablescan: 2 TU (time unit)
    - Selecting (testing) one tuple: 1 TU
    - Projecting one tuple: 1 TU
  - **Pipelined**
    - Operations do not create intermediate results
    - Each finished tuple is fed directly into the next operation
    - Tuples “flow” through the operations
  - **BNL Join**
    - Table scan, selection, projection
    - 1,000 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
  - **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 w.r.t. response time
      - First tuple arrives at 3100 ≪ t ≤ 13100

3.3 Response Time Models

- 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 w.r.t. response time
    - First tuple arrives at 3100 ≪ t ≤ 13100
### 3.3 Response Time Models

- **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

- **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 $x$ (via pipeline) of $A$ is hashed by join attribute
    - Compare $x$ to each tuple in the respective $B$ bucket
      - Return those tuples which show matching join attributes
      - If a new tuple arrives, process it analogously

- **Double-Pipelined-Hash-Join**
  - Dynamically build a hashtable for $A$ and $B$ each
    - Memory intensive!
  - Process tuples on arrival
    - Cache tuples if necessary
    - Balances between $A$ and $B$ tuples for better performance
    - Rely on statistics for a good $A:B$ ratio
  - If a new $A$ tuple arrives
    - Insert 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

- **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 $= $ Minimal Message Size
    - Receiving & decoding message
    - Closing channel

- **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
**Distributed Query Processing**

- **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

**Distributed Query Processing**

- 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**

**Next Lecture**

- **Distributed Transaction Management**
  - Transaction Synchronization
  - Distributed Two-Phase Commits
  - Byzantine Agreements