CMSC 724: Database Management Systems
Query Processing and Optimization

Instructor: Amol Deshpande
amol@cs.umd.edu
Outline

- **Part 1 Slides**
  - Query evaluation techniques for large databases, Skew Avoidance, Query compilation/vectorization
  - Query Optimization: Overview, How good are the query optimizers, really?, Reordering for Outerjoins, Query Rewriting

- **Adaptive Query Processing**
  - Eddies
  - Progressive Query Optimization
  - Compilation and adaptivity

- **Worst case optimal joins**

- **Froid: Databases and UDFs**
Traditional Optimization not Robust Enough

- In traditional settings:
  - Queries over many tables
  - Unreliability of traditional cost estimation
  - Success, maturity make problems more apparent, critical

- In new environments:
  - e.g. data integration, web services, streams, P2P...
  - Unknown dynamic characteristics for data and runtime
  - Increasingly aggressive sharing of resources and computation
  - Interactivity in query processing

- Note two distinct themes lead to the same conclusion:
  - Unknowns: even static properties often unknown in new environments and often unknowable a priori
  - Dynamics: environment changes can be very high

- Motivates intra-query adaptivity
Some Related Topics

- Autonomic/self-tuning optimization
  - Chen and Roussopoulos: Adaptive selectivity estimation [SIGMOD 1994]
  - LEO (@IBM), SITS (@MSR): Learning from previous executions

- Robust/least-expected cost optimization

- Parametric optimization
  - Choose a collection of plans, each optimal for a different setting of parameters
  - Select one at the beginning of execution

- Competitive optimization
  - Start off multiple plans... kill all but one after a while

- Adaptive operators
AQP: Overview/Summary

- Low-overhead, evolutionary approaches
  - Typically apply to non-pipelined execution
  - **Late binding**: Don’t instantiate the entire plan at start
  - **Mid-query reoptimization**: At “materialization” points, review the remaining plan and possibly re-optimize

- Pipelined execution
  - No materialization points, so the above doesn’t apply
  - The operators may contain complex states, raising correctness issues
  - **Eddies**
    - Always guarantee correct execution, but allows reordering during execution

- Lot of work in 1998-2008 timeframe -- not much since
AQP: Overview/Summary

- We will start with a general overview of AQP as presented in a later survey and tutorial.

- Then go through the three papers (first two quickly, and the last one in more detail):
  - First two will be covered in the tutorial.
Adaptive Query Processing Tutorial
VLDB 2008

Amol Deshpande, University of Maryland
Zachary G. Ives, University of Pennsylvania
Vijayshankar Raman, IBM Almaden Research Center

Thanks to Joseph M. Hellerstein, University of California, Berkeley
Query Processing: Adapting to the World

Data independence facilitates modern DBMS technology
- Separates specification ("what") from implementation ("how")
- Optimizer maps declarative query $\Rightarrow$ algebraic operations

Platforms, conditions are constantly changing:
$$\frac{d\text{app}}{dt} \ll \frac{d\text{env}}{dt}$$

Query processing **adapts** implementation to runtime conditions
- Static applications $\Rightarrow$ dynamic environments
Query Optimization and Processing
(As Established in System R [SAC+’79])

> UPDATE STATISTICS

Professor Course Student

> SELECT *
FROM Professor P,
Course C, Student S
WHERE P.pid = C.pid
AND S.sid = C.sid

Dynamic Programming + Pruning Heuristics
Traditional Optimization Is Breaking

In traditional settings:
- Queries over many tables
- Unreliability of traditional cost estimation
- Success & maturity make problems more apparent, critical

In new environments:
- e.g. data integration, web services, streams, P2P, sensor nets, hosting
- Unknown and dynamic characteristics for data and runtime
- Increasingly aggressive sharing of resources and computation
- Interactivity in query processing

Note two distinct themes lead to the same conclusion:
- Unknowns: even static properties often unknown in new environments and often unknowable a priori
- Dynamics: $\frac{denv}{dt}$ can be very high

Motivates intra-query adaptivity
A Call for Greater Adaptivity

System R adapted query processing as stats were updated

– Measurement/analysis: periodic
– Planning/actuation: once per query
– Improved thru the late 90s (see [Graefe '93] [Chaudhuri '98])
  Better measurement, models, search strategies

INGRES adapted execution many times per query

– Each tuple could join with relations in a different order
– Different plan space, overheads, frequency of adaptivity
  Didn’t match applications & performance at that time

Recent work considers adaptivity in new contexts
Tutorial Focus

By necessity, we will cover only a piece of the picture here

– Intra-query adaptivity:
  • autonomic / self-tuning optimization [CR’94, CN’97, BC’02, …]
  • robust / least expected cost optimization [CHG’02, MRS+’04, BC’05, …]
  • parametric or competitive optimization [A’93, INSS’92, CG’94, …]
  • adaptive operators, e.g., memory adaptive sort & hash join [NKT’88, KNT’89, PCL’93a, PCL’93b,…]

– Conventional relations, rather than streams
– Single-site, single query computation

- For more depth, see our survey in now Publishers’ *Foundations and Trends in Databases*, Vol. 1 No. 1
Tutorial Outline

- Motivation
- Non-pipelined execution
- Pipelined execution
  - Selection ordering
  - Multi-way join queries
- Putting it all in context
- Recap/open problems
Low-Overhead Adaptivity: Non-pipelined Execution
Late Binding; Staged Execution

Materialization points make natural decision points where the next stage can be changed with little cost:

- Re-run optimizer at each point to get the next stage
- Choose among precomputed set of plans – *parametric* query optimization [INSS’92, CG’94, …]

Normal execution: pipelines separated by materialization points
e.g., at a sort, GROUP BY, etc.
Mid-query Reoptimization

[ KD’98, MRS+04 ]

Choose **checkpoints** at which to monitor cardinalities

*Balance overhead and opportunities for switching plans*

If actual cardinality is **too different** from estimated,

*Avoid unnecessary plan re-optimization (where the plan doesn’t change)*

**Re-optimize** to switch to a new plan

*Try to maintain previous computation during plan switching*

- Most widely studied technique:
  - Federated systems (InterViso 90, MOOD 96), Red Brick,
  - Query scrambling (96), Mid-query re-optimization (98),
  - Progressive Optimization (04), Proactive Reoptimization (05), …
Mid-query Reoptimization

- At *materialization points*, re-evaluate the rest of the query plan
- Example:

  Initial query plan chosen

```
R.a = 10
R.b < 20
R.c like ...
```

Materialize R1

Estimated selectivities

0.05 0.1 0.2

A *free* opportunity to re-evaluate *the rest of the query plan*

- Exploit by gathering information about the materialized result
Mid-query Reoptimization

- At *materialization points*, re-evaluate the rest of the query plan
- Example:

```
R.a = 10
R.b < 20
R.c like ...
```

```
Materialize R1; build 1-d hists
```

Initial query plan chosen

- Estimated selectivities
  - R1: 0.05
  - R2: 0.1
  - R3: 0.2

A *free* opportunity to re-evaluate *the rest of the query plan*
- Exploit by gathering information about the materialized result
Mid-query Reoptimization

- At *materialization points*, re-evaluate the rest of the query plan.
- Example:

```
@materialization point

Example:
```

Initial query plan chosen

Estimated selectivities

Re-estimated selectivities

Significantly different \(\rightarrow\) original plan probably sub-optimal
Reoptimize the remaining part of the query
Where to Place Checkpoints?

Lazy checkpoints: placed above materialization points
- No work need be wasted if we switch plans here

Eager checkpoints: can be placed anywhere
- May have to discard some partially computed results
- Useful where optimizer estimates have high uncertainty

More checkpoints $\Rightarrow$ more opportunities for switching plans
Overhead of (simple) monitoring is small

Consideration: it is easier to switch plans at some checkpoints than others

Lazy checkpoints: placed above materialization points

Eager checkpoints: can be placed anywhere
When to Re-optimize?

- Suppose actual cardinality is different from estimates: how high a difference should trigger a re-optimization?

- Idea: do not re-optimize if current plan is still the best

1. Heuristics-based [KD’98]:
   - e.g., re-optimize < time to finish execution

2. Validity range [MRS+04]: precomputed range of a parameter (e.g., a cardinality) within which plan is optimal
   - Place eager checkpoints where the validity range is narrow
   - Re-optimize if value falls outside this range
   - Variation: bounding boxes [BBD’05]
How to Reoptimize

Getting a better plan:
- Plug in actual cardinality information acquired during this query (as possibly histograms), and re-run the optimizer

Reusing work when switching to the better plan:
- Treat fully computed intermediate results as materialized views
  - Everything that is under a materialization point
- Note: It is optional for the optimizer to use these in the new plan

Other approaches are possible (e.g., query scrambling [UFA’98])
Pipelined Execution
Adapting pipelined execution is often necessary:

- Too few materializations in today’s systems
- Long-running queries
- Wide-area data sources
- Potentially endless data streams

The tricky issues:

- Some results may have been delivered to the user
  - Ensuring correctness non-trivial
- Database operators build up *state*
  - Must reason about it during adaptation
  - May need to manipulate state
Adapting Pipelined Queries

We discuss three subclasses of the problem:

- **Selection ordering (stateless)**
  - Very good analytical and theoretical results
  - Increasingly important in web querying, streams, sensornets
  - Certain classes of join queries reduce to them

- **Select-project-join queries (stateful)**
  - *History-independent* execution
    - Operator state largely independent of execution history
    - Execution decisions for a tuple independent of prior tuples
  - *History-dependent* execution
    - Operator state depends on execution history
    - Must reason about the state during adaptation
Pipelined Execution Part I: Adaptive Selection Ordering
Adaptive Selection Ordering

Complex predicates on single relations common
  - e.g., on an employee relation:
    $((\text{salary} > 120000) \land (\text{status} = 2)) \lor$
    $((\text{salary} \text{ between } 90000 \text{ and } 120000) \land (\text{age} < 30) \land (\text{status} = 1)) \lor \ldots$

Selection ordering problem:

*Decide the order in which to evaluate the individual predicates against the tuples*

We focus on *conjunctive predicates* (containing only AND’s)

Example Query

```
select * from R
where R.a = 10 and R.b < 20
and R.c like '%name%';
```
Find a *single order of the selections* to be used for *all tuples*

**Query**

```sql
select * from R
where R.a = 10 and R.b < 20
and R.c like '%name%';
```

**Query plans considered**

- 3! = 6 distinct plans possible
Static Optimization

Cost metric: CPU instructions

Computing the cost of a plan

- Need to know the \textit{costs} and the \textit{selectivities} of the predicates

\[
\text{cost(plan)} = |R| \times (c_1 + s_1 \times c_2 + s_1 \times s_2 \times c_3)
\]
Static Optimization

**Rank ordering algorithm for independent selections** [IK’84]

- Apply the predicates in the decreasing order of rank:
  
  \[(1 - s) / c\]

  where \(s = \) selectivity, \(c = \) cost

For **correlated** selections:

- NP-hard under several different formulations
  
  • e.g. when given a random sample of the relation

- Greedy algorithm, shown to be 4-approximate [BMMNW’04]:
  
  • Apply the selection with the highest \((1 - s)/c\)
  
  • Compute the selectivities of remaining selections over the *result*
    
    • *Conditional selectivities*
  
  • Repeat
Query processing as routing of tuples through operators

A traditional pipelined query plan

Pipelined query execution using an eddy

An eddy operator
- Intercepts tuples from sources and output tuples from operators
- Executes query by routing source tuples through operators

Encapsulates all aspects of adaptivity in a “standard” dataflow operator: measure, model, plan and actuate.
Eddies [AH’00]

An R Tuple: \( r1 \)

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>10</td>
<td>AnameA</td>
<td>...</td>
</tr>
</tbody>
</table>

- \( R \) \( \xrightarrow{r1} \) Eddy
- \( R.a = 10 \)
- \( R.b < 20 \)
- \( R.c \) like ...
- \( \text{result} \)
Eddies [AH’00]

An R Tuple: \( r_1 \)

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>...</th>
<th>ready</th>
<th>done</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>10</td>
<td>AnameA</td>
<td>...</td>
<td>111</td>
<td>000</td>
</tr>
</tbody>
</table>

**ready bit** \( i \):
1 → operator \( i \) can be applied
0 → operator \( i \) can’t be applied

**Eddy**
\( R.a = 10 \)
\( R.b < 20 \)
\( R.c \) like …

Operator 1
Operator 2
Operator 3
An R Tuple: \( r1 \)

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>...</th>
<th>ready</th>
<th>done</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>10</td>
<td>AnnameA</td>
<td>...</td>
<td>111</td>
<td>000</td>
</tr>
</tbody>
</table>

\textit{done bit i:}

1 \( \rightarrow \) operator i has been applied
0 \( \rightarrow \) operator i hasn’t been applied

Eddies [AH’00]

\( R.\text{a} = 10 \)

\( R.\text{b} < 20 \)

\( R.\text{c} \text{ like } \ldots \)
An R Tuple: \( r_1 \)

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>...</th>
<th>ready</th>
<th>done</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>10</td>
<td>AnameA</td>
<td>...</td>
<td>111</td>
<td>000</td>
</tr>
</tbody>
</table>

Used to decide validity and need of applying operators

Eddy

\( R.a = 10 \)

\( R.b < 20 \)

\( R.c \) like ...

result
An R Tuple: \( r_1 \)

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>...</th>
<th>ready</th>
<th>done</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>10</td>
<td>AnameA</td>
<td>...</td>
<td>101</td>
<td>000</td>
</tr>
</tbody>
</table>

Eddy looks at the next tuple

For a query with only selections, \( \text{ready} = \text{complement(} \text{done} \text{)} \)

Operator 1: \( R.a = 10 \)
Operator 2: \( R.b < 20 \)
Operator 3: \( R.c \text{ like } \ldots \)

\text{not satisfied}
Eddies [AH’00]

An R Tuple: \( r_2 \)

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>15</td>
<td>AnnameA</td>
<td>...</td>
</tr>
</tbody>
</table>

- \( R.a = 10 \)
- \( R.b < 20 \)
- \( R.c \text{ like ...} \)

\( R \rightarrow r_2 \) → Eddy → \( \text{satisfied} \)

Operator 1
- \( R.a = 10 \) → \( \text{satisfied} \)

Operator 2
- \( R.b < 20 \) → \( \text{satisfied} \)

Operator 3
- \( R.c \text{ like ...} \) → \( \text{satisfied} \)

result
Eddies [AH’00]

An R Tuple: \( r2 \)

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>...</th>
<th>ready</th>
<th>done</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>15</td>
<td>AnnameA</td>
<td>...</td>
<td>000</td>
<td>111</td>
</tr>
</tbody>
</table>

If \( \text{done} = 111 \), send to output

Operator 1:
- \( R.a = 10 \)

Operator 2:
- \( R.b < 20 \)

Operator 3:
- \( R.c \) like ...

Result: \( r2 \)
Adapting order is easy
– Just change the operators to which tuples are sent
– Can be done on a per-tuple basis
– Can be done in the middle of tuple’s “pipeline”

How are the routing decisions made?
Using a routing policy
Routing Policies that Have Been Studied

**Deterministic** [D03]
- Monitor costs & selectivities continuously
- Re-optimize periodically using rank ordering (or A-Greedy for correlated predicates)

**Lottery scheduling** [AH00]
- Each operator runs in thread with an input queue
- “Tickets” assigned according to tuples input / output
- Route tuple to next eligible operator with room in queue, based on number of “tickets” and “backpressure”

**Content-based routing** [BBDW05]
- Different routes for different plans based on attribute values
Routing Policy 1: Non-adaptive

- Simulating a single static order
  - E.g. operator 1, then operator 2, then operator 3

Routing policy:

```
if done =
000  \rightarrow \text{route to 1}
100  \rightarrow \text{route to 2}
110  \rightarrow \text{route to 3}
```

Table lookups $\rightarrow$ very efficient
Overhead of Routing

- PostgreSQL implementation of eddies using *bitset lookups* [Telegraph Project]
- Queries with 3 selections, of varying cost
  - Routing policy uses a *single static order*, i.e., no adaptation
Routing Policy 2: Deterministic

- Monitor costs and selectivities *continuously*
- Reoptimize *periodically* using KBZ

**Statistics Maintained:**
- Costs of operators
- Selectivities of operators

**Routing policy:**
- Use a single order for a batch of tuples
- Periodically apply KBZ

Can use specialized policies for correlated predicates

- Operator 1
  - $R.a = 10$

- Operator 2
  - $R.b < 20$

- Operator 3
  - $R.c$ like …

result
Adaptation using *batching*
- Reoptimized every $X$ tuples using monitored selectivities
- Identical selectivities throughout $\Rightarrow$ experiment measures only the overhead
Routing Policy 3: Lottery Scheduling

- Originally suggested routing policy [AH’00]
- Applicable only if each operator runs in a separate thread
- Uses two easily obtainable pieces of information for making routing decisions:
  - Busy/idle status of operators
  - Tickets per operator
Routing Policy 3: Lottery Scheduling

- Routing decisions based on busy/idle status of operators

**Rule:**
IF operator busy, THEN do not route more tuples to it

**Rationale:**
Every thread gets equal time
SO IF an operator is busy, THEN its cost is perhaps very high
Routing Policy 3: Lottery Scheduling

- Routing decisions based on tickets

**Rules:**
1. Route a new tuple randomly weighted according to the number of tickets

<table>
<thead>
<tr>
<th>Operator</th>
<th>Tickets</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>O1</td>
<td>10</td>
<td>0.1</td>
</tr>
<tr>
<td>O2</td>
<td>70</td>
<td>0.7</td>
</tr>
<tr>
<td>O3</td>
<td>20</td>
<td>0.2</td>
</tr>
</tbody>
</table>

Will be routed to:
- O1 w.p. 0.1
- O2 w.p. 0.7
- O3 w.p. 0.2
Routing Policy 3: Lottery Scheduling

- Routing decisions based on tickets

**Rules:**
1. Route a new tuple randomly weighted according to the number of tickets

<table>
<thead>
<tr>
<th>Operator 1</th>
<th>Operator 2</th>
<th>Operator 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>R.a = 10</td>
<td>R.b &lt; 20</td>
<td>R.c like ...</td>
</tr>
</tbody>
</table>

tickets(O1) = 10  
tickets(O2) = 70  
tickets(O3) = 20
Routing Policy 3: Lottery Scheduling

- Routing decisions based on tickets

**Rules:**
1. Route a new tuple randomly weighted according to the number of tickets
2. Route a tuple to an operator \( O_i \)

\[ \text{tickets}(O_1) = 11 \]
\[ \text{tickets}(O_2) = 70 \]
\[ \text{tickets}(O_3) = 20 \]
Routing Policy 3: Lottery Scheduling

- Routing decisions based on tickets

**Rules:**
1. Route a new tuple randomly weighted according to the number of tickets
2. Route a tuple to an operator \( O_i \)
   
   \[ \text{tickets}(O_i) ++; \]
3. \( O_i \) returns a tuple to eddy
   
   \[ \text{tickets}(O_i) --; \]

- tickets\((O1) = 11\)
- tickets\((O2) = 70\)
- tickets\((O3) = 20\)
Routing Policy 3: Lottery Scheduling

- Routing decisions based on tickets

*Rules:*
1. Route a new tuple randomly weighted according to the number of tickets
2. Route a tuple to an operator $O_i$ 
   
   $\text{tickets}(O_i) ++$;
3. $O_i$ returns a tuple to eddy
   
   $\text{tickets}(O_i) --$;

Will be routed to:
- $O_2$ w.p. 0.777
- $O_3$ w.p. 0.222

$tickets(O_1) = 10$
$tickets(O_2) = 70$
$tickets(O_3) = 20$
Routing Policy 3: Lottery Scheduling

- Routing decisions based on tickets

**Rules:**
1. Route a new tuple randomly weighted according to the number of tickets
2. Route a tuple to an operator $O_i$
   - tickets($O_i$) ++;
3. $O_i$ returns a tuple to eddy
   - tickets($O_i$) --;

**Rationale:**
- Tickets($O_i$) roughly corresponds to $(1 - \text{selectivity}(O_i))$
- So more tuples are routed to highly selective operators

tickets($O1$) = 10
tickets($O2$) = 70
tickets($O3$) = 20
Routing Policy 3: Lottery Scheduling

- Effect of the combined lottery scheduling policy:
  - Low cost operators get more tuples
  - Highly selective operators get more tuples
  - Some tuples are knowingly routed according to sub-optimal orders
    - To explore
    - Necessary to detect selectivity changes over time
Plan Space explored
- Allows arbitrary “horizontal partitioning”
- Not necessarily correlated with order of arrival

In a later paper, we looked at optimizing for horizontal partitioning directly
Pipelined Execution Part II: Adaptive Join Processing
Adaptive Join Processing: Outline

- Single streaming relation
  - Left-deep pipelined plans
- Multiple streaming relations
  - Execution strategies for multi-way joins
  - History-independent execution
  - History-dependent execution
Simplest method of joining tables
- Pick a **driver** table (R). Call the rest **driven** tables
- Pick access methods (AMs) on the driven tables (*scan*, *hash*, or *index*)
- Order the driven tables
- Flow R tuples through the driven tables

For each $r \in R$ do:
look for matches for $r$ in A;
for each match $a$ do:
   look for matches for <$r,a$> in B;
   ...

Adapting a Left-deep Pipelined Plan

Simplest method of joining tables
- Pick a *driver* table (R). Call the rest *driven* tables
- Pick access methods (AMs) on the driven tables
- Order the driven tables
- Flow R tuples through the driven tables

For each \( r \in R \) do:
look for matches for \( r \) in A;
for each match a do:
look for matches for \( <r,a> \) in B;
...

Almost identical to selection ordering
Adapting the Join Order

Let $c_i =$ cost/lookup into $i$’th driven table, $s_i =$ fanout of the lookup

As with selection, cost = $|R| \times (c_1 + s_1 c_2 + s_1 s_2 c_3)$

Caveats:
- Fanouts $s_1, s_2, \ldots$ can be > 1
- Precedence constraints
- Caching issues

Can use rank ordering, A-greedy for adaptation (subject to the caveats)
Adapting a Left-deep Pipelined Plan

Simplest method of joining tables
- Pick a *driver* table (R). Call the rest *driven* tables
- Pick access methods (AMs) on the driven tables
- Order the driven tables
- Flow R tuples through the driven tables

For each \( r \in R \) do:
look for matches for \( r \) in A;
for each match a do:
  look for matches for \(<r,a>\) in B;
  ...

Adapting a Left-deep Pipelined Plan

Key issue: Duplicates
Adapting the choice of driver table
  [L+07] Carefully use indexes to achieve this
Adapting the choice of access methods
  – Static optimization: explore all possibilities and pick best
  – Adaptive: Run multiple plans in parallel for a while, and then pick one and discard the rest [Antoshenko’s 96]
    • Cannot easily explore combinatorial options
Adaptive Join Processing: Outline

- Single streaming relation
  - Left-deep pipelined plans

- Multiple streaming relations
  - Execution strategies for multi-way joins
  - History-independent execution
    - MJoins
  - History-dependent execution
    - Eddies with joins
    - Corrective query processing
**Example Join Query & Database**

```sql
select *
from students, enrolled, courses
where students.name = enrolled.name
and enrolled.course = courses.course
```
Symmetric/Pipelined Hash Join
[RS86, WA91]

```
select * from students, enrolled where students.name = enrolled.name
```

- Simultaneously builds and probes hash tables on both sides
- Widely used:
  - adaptive query processing
  - stream joins
  - online aggregation
  - ...
- Naïve version degrades to NLJ once memory runs out
  - Quadratic time complexity
  - memory needed = sum of inputs
- Improved by XJoins [UF 00], Tukwila DPJ [IFFLW 99]
Multi-way Pipelined Joins over Streaming Relations

Alternatives

- Using binary join operators

- Using a single n-ary join operator (MJoin) [VNB’03]

- Some other options explored in the literature
**Materialized state that depends on the query plan used**

*History-dependent!*
Multi-way Pipelined Joins over Streaming Relations

Three alternatives

- Using binary join operators
  - History-dependent execution
  - Hard to reason about the impact of adaptation
  - May need to migrate the state when changing plans

- Using a single n-ary join operator (MJoin) [VNB’03]
Probing Sequences

Students tuple: Enrolled, then Courses
Enrolled tuple: Students, then Courses
Courses tuple: Enrolled, then Students

Hash tables contain all tuples that arrived so far
Irrespective of the probing sequences used

History-independent execution!
MJoins [VNB’03]

Choosing probing sequences

– For each relation, use a left-deep pipelined plan (based on hash indexes)

– Can use selection ordering algorithms
  Independently for each relation

Adapting MJoins

– Adapt each probing sequence independently
  e.g., StreaMon [BW’01] used A-Greedy for this purpose

A-Caching [BMWM’05]

– Maintain intermediate caches to avoid recomputation
– Alleviates some of the performance concerns
Adaptive Join Processing: Outline

- Single streaming relation
  - Left-deep pipelined plans
- Multiple streaming relations
  - Execution strategies for multi-way joins
  - History-independent execution
    - MJoins
    - SteMs
  - History-dependent execution
    - Eddies with binary joins
Eddies with Binary Joins [AH’00]

For correctness, must obey routing constraints!!
Eddies with Binary Joins [AH’00]

For correctness, must obey routing constraints!!
Eddies with Binary Joins [AH’00]

For correctness, must obey routing constraints!!
Use some form of tuple-lineage
Eddies with Binary Joins [AH’00]

Can use any join algorithms
But, pipelined operators preferred
Provide quick feedback
Eddies with Symmetric Hash Joins
As a result of routing decisions, state gets embedded inside the operators.
Recap: Eddies with Binary Joins

Routing constraints enforced using tuple-level lineage

Must choose access methods, join spanning tree beforehand
  – SteMs relax this restriction [RDH’03]

The operator state makes the behavior unpredictable
  – Unless only one streaming relation

Routing policies explored are same as for selections
  – Can tune policy for interactivity metric [RH’02]
Outline

- Query evaluation techniques for large databases, Skew Avoidance, Query compilation/vectorization
- Query Optimization: Overview, How good are the query optimizers, really?, Reordering for Outerjoins, Query Rewriting
- Adaptive Query Processing
  - Eddies
  - Progressive Query Optimization
  - Compilation and adaptivity
Overview

- Continuously "reorder" operators as the query is executing
  - By changing the "order" in which tuples visit operators
  - Obviate the need for selectivity estimation and optimization entirely
  - Naturally handles situations where the selectivities change over time (for long-running queries)
Eddies and Joins

- Selections are arbitrarily reorderable
- What about joins?

- These two are tricky
- Nested loops requires iterating over all of inner
- Hash join requires building a hash table on inner

- An index lookup can be treated as a “selection”
- Send an S tuple, get back augmented tuples
- Note: decision to use the index cannot be “adapted”
Reorderability of Plans

- **Synchronization Barriers**
  - Many operators explicitly enforce an order in which tuples must be read from the inputs
  - e.g., Sort-merge joins: at most points, the next tuple to read must be read from a specific input
  - Hash joins: need to read all of “inner” before outer tuples can be read

- **Moments of Symmetry**
  - Sort-merge join is symmetric
  - But Nested-loops is not
    - However, can change the outer/inner at specific points

- **Join operators with more moments of symmetric preferred**
  - e.g., Symmetric Hash Join Operator
Reorderability of Plans

Figure 3: Tuples generated by block, index, and hash ripple join. In block ripple, all tuples are generated by the join, but some may be eliminated by the join predicate. The arrows for index and hash ripple join represent the *logical* portion of the cross-product space checked so far; these joins only expend work on tuples satisfying the join predicate (black dots). In the hash ripple diagram, one relation arrives $3 \times$ faster than the other.
Eddies

- Implemented in the context of River project
- Eddy is a separate module that talks to all other operators
  - Uses “ready” and “done” bitsets to direct traffic
- Lottery scheduling-based routing policy
  - Promising initial results, but bunch of caveats
Query evaluation techniques for large databases, Skew Avoidance, Query compilation/vectorization

Query Optimization: Overview, How good are the query optimizers, really?, Reordering for Outerjoins, Query Rewriting

Adaptive Query Processing
  ◦ Eddies
  ◦ Progressive Query Optimization
  ◦ Compilation and adaptivity
Overview

- Trigger re-optimization during query execution if errors too high
- Through use of CHECK operators inserted into the query plan
  - Succeeds if the observed values within a range around the estimates
- If optimizer estimates accurate, the only overhead is the “counting” done by CHECK

![Diagram](image)

**Figure 2**: Adding CHECK to the outer of a NLJN
Overview

- Trigger re-optimization during query execution if errors too high
- Through use of CHECK operators inserted into the query plan
  - Succeeds if the observed values within a range around the estimates
- If optimizer estimates accurate, the only overhead is the “couting” done by CHECK

- If CHECK detects significant error, then “reoptimize”
  - Partial results made available to the optimizer to use if it wants (in the form of a materialized view)
when we are sure that the plan will change. In general, this is crucial to minimize risk.

Figure 1 distinguishes the architecture of a DBMS with POP capability. To illustrate those enhancements to the architecture of a data management system, as depicted in Figure 2, a more suitable join method such as hash join is compared to the actual cardinality observed by the runtime system. If the check range is violated, the runtime system retains intermediate results together with their actual cardinality values and triggers re-optimization of the query. Actual cardinalities measured during the initial run help the re-optimization step avoid the same mistake. After optimization run for a second re-optimization.

Figure 2

Adding logic to the post-pass of the optimizer for deciding the most judicious location of CHECKs. Adding logic to the plan generator of the query optimizer to determine the check range by determining which plan operator is optimal in the current plan. Adding logic to exploit intermediate results when adding logic to the runtime system for interpreting CHECK.

**Figure 1: Progressive Optimization architecture**
Computing Validity Ranges

- Helps only re-optimize when necessary

- The general problem is that of “parametric” optimization
  - i.e., find the best plan for each combination of parameters
  - Too expensive

Instead:

- Consider P1 and P2 -- two identical plans except for the top operator
- Let cost(P1) < cost(P2) per the estimates → we would choose P1 over P2
- Let “x” denote an edge into the top operator, and let “result(x) = e” denote the result flowing along “x”
- Figure out: at what value of |result(x)|, we would have chosen P2 instead

\[ \text{Input Cardinality} \]

\[ \begin{align*}
\text{Cost}(P_{\text{opt}}) & \quad \text{Cost}(P_{\text{alt}}) \\
\text{Cost}(P_{\text{opt}}) & \quad \text{Cost}(P_{\text{alt}})
\end{align*} \]

\[ \text{Input Cardinality} \]

\[ \text{Cost} \]

\[ e \quad c \]

\[ \text{Figure 4 illustrates how we can narrow the validity range of this edge. As we settle on the upper bound of the validity range of this edge. As we settle on the upper bound of the validity range of this edge. As we settle on the upper bound of the validity range of this edge. As we settle on the upper bound of the validity range of this edge. As we settle on the upper bound of the validity range of this edge. As we settle on the upper bound of the validity range of this edge. As we settle on the upper bound of the validity range of this edge. As we settle on the upper bound of the validity range of this edge. As we settle on the upper bound of the validity range of this edge. As we settle on the upper bound of the validity range of this edge. As we settle on the upper bound of the validity range of this edge. As we settle on the upper bound of the validity range of this edge. As we settle on the upper bound of the validity range of this edge. As we settle on the upper bound of the validity range of this edge. As we settle on the upper bound of the validity range of this edge. As we settle on the upper bound of the validity range of this edge. As we settle on the upper bound of the validity range of this edge. As we settle on the upper bound of the validity range of this edge. As we settle on the upper bound of the validity range of this edge. As we settle on the upper bound of the validity range of this edge. As we settle on the upper bound of the validity range of this edge. As we settle on the upper bound of the validity range of this edge. As we settle on the upper bound of the validity range of this edge. As we settle on the upper bound of the validity range of this edge.
\]
Computing Validity Ranges

- Helps only re-optimize when necessary
- The general problem is that of “parametric” optimization
  - i.e., find the best plan for each combination of parameters
  - Too expensive
- Instead:
  - Consider P1 and P2 -- two identical plans except for the top operator
  - Let cost(P1) < cost(P2) per the estimates → we would choose P1 over P2
  - Let “x” denote an edge into the top operator, and let “result(x) = e” denote the result flowing along “x”
  - Figure out: at what value of |result(x)|, we would have chosen P2 instead
- Use numerical techniques to find these validity ranges
Reusing Partial Results

- Treat it as a materialized view, and let the optimizer decide
- If the plan under CHECK has a side-effect (e.g., update), then must reuse that plan (i.e., not redo that portion)
- In many cases, better not to use the partial result

Figure 6: Two alternatives considered in re-optimization
Lazy vs Eager Checking

- If there is already a materialization point, can add CHECK there for free (lazy)
- Can add explicit materialization along with a CHECK
  - Extra overhead in doing that
- Eager CHECKs don’t wait for materialization
- ECWC (Eager without compensation)
  - There is a materialization afterwards → no results will be output to the user
  - So can easily reoptimize without worrying about compensation

**Figure 7**: Lazy checking (LC) and eager checking without compensation (ECWC)
Eager Checking

- With Buffering: Buffer results until you are sure things are okay
  - Delays the pipeline for some time

**Figure 8**: Eager checking with Buffering
Eager Checking

- With Deferred Compensation
  - Keep track of what tuples have already been output
  - Check that side table before outputting new tuples after reoptimization
  - Potentially a lot of repeated work

![Diagram showing Eager Checking with deferred compensation and re-optimization.]
Experiments

- Degradation in some cases -- sometimes two errors cancelled each other out in the original plan

![Scatter Plot of Response Times with and without POP on the DMV database](image1)

**Figure 15:** Scatter Plot of Response Times with and without POP on the DMV database

![Speedup and Regression of each Query](image2)

**Figure 16:** Speedup and Regression of each Query
Outline

- Query evaluation techniques for large databases, Skew Avoidance, Query compilation/vectorization
- Query Optimization: Overview, How good are the query optimizers, really?, Reordering for Outerjoins, Query Rewriting
- Adaptive Query Processing
  - Eddies
  - Progressive Query Optimization
  - Compilation and adaptivity
Adaptive query processing (POP-style) works well with interpretable query plans, but not as well with compilation

- Compiling a new query plan too expensive

Figure 1: Reoptimizing Compiled Queries – PCQ enables near-optimal execution through adaptivity with minimal compilation overhead.
Adaptive query processing (POP-style) works well with interpretable query plans, but not as well with compilation
  ◦ Compiling a new query plan too expensive

Instead:
  ◦ Precompile a bunch of different plans at optimization time itself
  ◦ Add indirections to the compiled code to make it easy to switch/permute operators
  ◦ Add hooks for collecting runtime performance metrics
    • To be used to decide whether to switch
Permutable Compiled Queries (PCQ)

Figure 2: System Overview – The DBMS translates the SQL query into a DSL that contains indirection layers to enable permutability. Next, the system compiles the DSL into a compact bytecode representation. Lastly, an interpreter executes the bytecode. During execution, the DBMS collects statistics for each predicate, analyzes this information, and permutes the ordering to improve performance.
Adaptive Filter Ordering

(a) Example Input SQL Query

```
SELECT * FROM A WHERE col1 * 3 = col2 + col3 AND col4 < 44
```

(b) Generated Code and Execution of Permutable Filter

```
1 fun query() {
2   var filters=[{p1,p2}]
3   for (v in A) {
4   }

6 fun p1(v:*Vec) {
7   @selectLT(v.col4,44)}

8 fun p2(v:*Vec) {
9   for (t in v) {
10      if (t.col1*3 ==
11        t.col2+t.col3){
12        v[t]=true}}

fun agg() {
   var rank = 0.8
   if (rank > 0.5)
   {
      p1 = rank
      p2 = 1 - rank
   }
   AggregateAggregation engine = new AggregateAggregation();
   engine.run();
}
```

Figure 3: Filter Reordering – The Translator converts the query in (a) into the TPL on the left side of (b). This program uses a data structure template with query-specific filter logic for each filter clause. The right side of (b) shows how the policy collects metrics and then permutes the ordering.
Adaptive Aggregations

(a) Example Input SQL Query

```
SELECT col1, COUNT(*) FROM A GROUP BY col1
```

(b) Generated Code and Execution of Adaptive Aggregation

Figure 4: Adaptive Aggregations – The input query in (a) is translated into TPL on the left side of (b). The right side of (b) steps through one execution of PCQ aggregation.
Adaptive Joins

(a) Example Input SQL Query

```
SELECT * FROM A
INNER JOIN B ON A.col1 = B.col1
INNER JOIN C ON A.col2 = C.col1
```

(b) Possible Join Orderings

(c) Generated Code and Execution of Permutable Joins

```
1 fun query() {
2   // HT on B, C built.
3   var joinExec = {{
4     {ht_B, joinB},
5     {ht_C, joinC}}
6   for (v in A) {
7     
8   }
9   fun joinB(
10      v:*Vec,m: [*]Entry)
11     for (t in v){
12       if (t.col1==m[t].col1){
13           v[t]=true}
14   }
15   fun joinC(
16      v:*Vec,m: [*]Entry) {
17       gatherSelectEq(v.col2, m, 0)}
```

Figure 5: Adaptive Joins – The DBMS translates the query in (a) to the program in (c). The right side of (c) illustrates one execution of a permutable join that includes a metric collection step.
We now present an analysis of the PCQ method and correspondingly implemented our microbenchmarks using the Google Benchmark library. The database consists of compiled queries. We execute these queries using a mix of left-deep, right-deep, and permutable join pipelines. We then measure PCQ's ability to improve the performance of these queries.

Figure 6: Performance Over Time - Execution time of three static filter orderings and our PCQ filter during a sequential table scan.

Figure 12: Varying Number of Joins – Execution time to perform a multi-step join while keeping the overall join selectivity at 10%.
Adaptivity Loop

**Measure what?**
Cardinalities/selectivities, operator costs, resource utilization

**Measure when?**
Continuously (eddies); using a random sample (A-greedy); at materialization points (mid-query reoptimization)

**Measurement overhead?**
Simple counter increments (mid-query) to very high
Adaptivity Loop

Analyze/replan what decisions?
(Assess actual vs. estimated selectivities)
Evaluate costs of alternatives and switching (keep state in mind)

Analyze / replan when?
Periodically; at materializations (mid-query); at conditions (A-greedy)

Plan how far ahead?
Next tuple; batch; next stage (staged); possible remainder of plan (CQP)

Planning overhead?
Switch stmt (parametric) to dynamic programming (CQP, mid-query)
Actuation: How do they switch to the new plan/new routing strategy?

Actuation overhead?

At the end of pipelines $\rightarrow$ free (mid-query)
During pipelines:
    History-independent $\rightarrow$ Essentially free (selections, MJoins)
    History-dependent $\rightarrow$ May need to migrate state (STAIRs, CAPE)
Recap/Thoughts

- Not much work on adaptive query processing in the last 10 years
  - SkinnerDB [2019] another relevant work

- More work on adapting the execution of a single operator
  - e.g., changing things based on available resources

- Likely to re-emerge as an important topic in the next few years
  - As QP in many systems becomes more mature...
  - As SQL starts becoming more and more common as the query language (e.g., in Spark, Pandas, etc).
Outline

- Query evaluation techniques for large databases, Skew Avoidance, Query compilation/vectorization
- Query Optimization: Overview, How good are the query optimizers, really?, Reordering for Outerjoins, Query Rewriting
- Adaptive Query Processing
- Worst-case Optimal Join Processing
- Froid: UDFs and Databases
Consider an “edges” relation with N edges, capturing an “undirected” graph,

And a query to find the number of “triangles”

<table>
<thead>
<tr>
<th>source</th>
<th>target</th>
</tr>
</thead>
<tbody>
<tr>
<td>v1</td>
<td>v2</td>
</tr>
<tr>
<td>v2</td>
<td>v1</td>
</tr>
<tr>
<td>v1</td>
<td>v3</td>
</tr>
<tr>
<td>v3</td>
<td>v1</td>
</tr>
<tr>
<td>v2</td>
<td>v3</td>
</tr>
<tr>
<td>v3</td>
<td>v2</td>
</tr>
</tbody>
</table>

select count(*)/6
from edges e1, edges e2, edges e3
where e1.target = e2.source and e2.target = e3.source and e3.target = e1.source

Any “binary joins” plan will be “sub-optimal”
Worst case = O(N^2)
However, output size bounded by O(N^1.5)
Yannakakis Algorithm [1981]

q() :- R(A, B), S(B, C), T(C, D)

Boolean Conjunctive Query
Answer is a True/False

However: No results in the output
### Yannakakis Algorithm [1981]

$q() \ :- \ R(A, B), S(B, C), T(C, D)$

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1</td>
<td>b1</td>
</tr>
<tr>
<td>a2</td>
<td>b1</td>
</tr>
<tr>
<td>a3</td>
<td>b1</td>
</tr>
<tr>
<td>a4</td>
<td>b1</td>
</tr>
<tr>
<td>a5</td>
<td>b1</td>
</tr>
<tr>
<td>a6</td>
<td>b1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>b1</td>
<td>c1</td>
</tr>
<tr>
<td>b1</td>
<td>c2</td>
</tr>
<tr>
<td>b1</td>
<td>c3</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>b2</td>
<td>c0</td>
</tr>
<tr>
<td>b3</td>
<td>c0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>c0</td>
<td>d1</td>
</tr>
<tr>
<td>c0</td>
<td>d2</td>
</tr>
<tr>
<td>c0</td>
<td>d3</td>
</tr>
<tr>
<td>c0</td>
<td>d4</td>
</tr>
<tr>
<td>c0</td>
<td>d5</td>
</tr>
<tr>
<td>c0</td>
<td>d6</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

No Binary Join Tree Works

- $R \ JOIN \ S \ == \ generates \ 1 \ trillion \ tuples$ (none of which match $T$)
- $S \ JOIN \ T \ == \ generates \ 1T \ tuples$
- $R \ JOIN \ T \ == \ cross \ product \ == \ 1T \ tuples$

1M tuples with $B = b1$
1M tuples with $C = c0$
1M tuples with $C = c0$
1M tuples with $B = b1$
Yannakakis Algorithm [1981]

q() :- R(A, B), S(B, C), T(C, D)

First, do S SEMIJOIN R

Removes tuples from S that don’t contribute to the final output (e.g., (b2, c0) will never join with anything from R)

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1</td>
<td>b1</td>
</tr>
<tr>
<td>a2</td>
<td>b1</td>
</tr>
<tr>
<td>a3</td>
<td>b1</td>
</tr>
<tr>
<td>a4</td>
<td>b1</td>
</tr>
<tr>
<td>a5</td>
<td>b1</td>
</tr>
<tr>
<td>a6</td>
<td>b1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>b1</td>
<td>c1</td>
</tr>
<tr>
<td>b1</td>
<td>c2</td>
</tr>
<tr>
<td>b1</td>
<td>c3</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>b2</td>
<td>c0</td>
</tr>
<tr>
<td>b3</td>
<td>c0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>c0</td>
<td>d1</td>
</tr>
<tr>
<td>c0</td>
<td>d2</td>
</tr>
<tr>
<td>c0</td>
<td>d3</td>
</tr>
<tr>
<td>c0</td>
<td>d4</td>
</tr>
<tr>
<td>c0</td>
<td>d5</td>
</tr>
<tr>
<td>c0</td>
<td>d6</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

1M tuples with B = b1
1M tuples with C = c0

1M tuples with C = c0
1M tuples with B = b1
Yannakakis Algorithm [1981]

q() :- R(A, B), S(B, C), T(C, D)

First, do S SEMIJOIN R

Then: X1 = T SEMIJOIN (S SEMIJOIN R)

Then, do X2 = S SEMIJOIN X1

To further “reduce” S by removing tuples that don’t join with anything from T

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1</td>
<td>b1</td>
</tr>
<tr>
<td>a2</td>
<td>b1</td>
</tr>
<tr>
<td>a3</td>
<td>b1</td>
</tr>
<tr>
<td>a4</td>
<td>b1</td>
</tr>
<tr>
<td>a5</td>
<td>b1</td>
</tr>
<tr>
<td>a6</td>
<td>b1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

1M tuples with B = b1

<table>
<thead>
<tr>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>b1</td>
<td>c1</td>
</tr>
<tr>
<td>b1</td>
<td>c2</td>
</tr>
<tr>
<td>b1</td>
<td>c3</td>
</tr>
<tr>
<td>b2</td>
<td>c0</td>
</tr>
<tr>
<td>b3</td>
<td>c0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

1M tuples with C = c0

<table>
<thead>
<tr>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>c0</td>
<td>d1</td>
</tr>
<tr>
<td>c0</td>
<td>d2</td>
</tr>
<tr>
<td>c0</td>
<td>d3</td>
</tr>
<tr>
<td>c0</td>
<td>d4</td>
</tr>
<tr>
<td>c0</td>
<td>d5</td>
</tr>
<tr>
<td>c0</td>
<td>d6</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

1M tuples with C = c0

<table>
<thead>
<tr>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

1M tuples with B = b1
Yannakakis Algorithm [1981]

\[ q() : R(A, B), S(B, C), T(C, D) \]

First, do \( S \text{ SEMIJOIN} R \)

Then: \( X_1 = T \text{ SEMIJOIN} (S \text{ SEMIJOIN} R) \)

Then, do \( X_2 = S \text{ SEMIJOIN} X_1 \)

Finally, do \( X_3 = R \text{ SEMIJOIN} X_2 \)
Yannakakis Algorithm [1981]

- Called “semi-join reducer sequences”
  - Basically get rid of tuples from each relation that don’t contribute to the output
  - Result EMPTY in our example, but in general, only relevant tuples will be left

- Once this is done, you can do join in any order
  - Guaranteed that the total time is “linear” in the total size of the inputs and output
  - Can’t avoid dependence on the output -- the join query may do a Cartesian product

- Can be generalized to any “acyclic” query
Acyclic Queries?

- Conjunctive queries as “hypergraphs”

\[ q() : \text{R1}(A, B, C), \text{R2}(B, C, D), \text{R3}(C, D, E) \]

Each attribute == a vertex
Each relation == a “hyperedge”
Conjunctive queries as “hypergraphs”

$q() : R1(A, B, C), R2(C, D, E), R3(A, E)$

Each attribute == a vertex
Each relation == a “hyperedge”
If all relations are 2 attributes, then the hypergraph is same as a graph

\[
q() \equiv R1(A, B), R2(B, C), R3(C, D), R4(D, A)
\]

Acyclic queries in this case == the graph has no cycles, i.e., the graph is a tree

More complex for hypergraphs
Structural Approaches

- For “acyclic” queries, can always find a semijoin reducer sequence
  - Can be done in optimal time: linear in size of inputs + output

- What about non-acyclic queries?
  - Try to define how “far” from acyclic-ness
  - Captured as ”width” of the hypergraph
    - Width of acyclic hypergraphs = 1

- AGM [FOCS, 2008] defined “fractional hypertree width”, and an algorithm that runs in \( O(N^{(fhw+1) \log N}) \)

- Several more practical algorithms since then, including one that was implemented before it was proved optimal
**Triangle Query**

\[ Q_\Delta = R(A, B) \bowtie S(B, C) \bowtie T(A, C). \]

\[ R = \{a_0\} \times \{b_0, \ldots, b_m\} \cup \{a_0, \ldots, a_m\} \times \{b_0\} \]

\[ S = \{b_0\} \times \{c_0, \ldots, c_m\} \cup \{b_0, \ldots, b_m\} \times \{c_0\} \]

\[ T = \{a_0\} \times \{c_0, \ldots, c_m\} \cup \{a_0, \ldots, a_m\} \times \{c_0\} \]

Each relation has: 2m + 1 tuples
Output = 3m + 1
Any pairwise join has size: m^2 + m
Projections/Semi-joins don’t help
Algorithm 1: Power of Two Choices

Skew in the relations: a_0 generates a lot of intermediate tuples, but not as many output tuples

\[ Q_\Delta[a_i] := \pi_B,C(\sigma_{A=a_i}(Q_\Delta)). \]

Call a_i heavy if:

\[ |\sigma_{A=a_i}(R \bowtie T)| \geq |Q_\Delta[a_i]|. \]

Two Choices for each a_i:

(i) Compute \( \sigma_{A=a_i}(R) \bowtie \sigma_{A=a_i}(T) \) and filter the results by probing against S or
(ii) Consider each tuple in \((b, c) \in S\) and check if \((a_i, b) \in R\) and \((a_i, c) \in T\).

Can prove to run in: \( O(N^{1.5}) \)
Algorithm 1: Power of Two Choices

Algorithm 1 Computing $Q_\Delta$ with power of two choices.

**Input:** $R(A, B), S(B, C), T(A, C)$ in sorted order

1. $Q_\Delta \leftarrow \emptyset$
2. $L \leftarrow \pi_A(R) \cap \pi_A(T)$
3. **For** each $a \in L$ **do**
   4. **If** $|\sigma_{A=a}R| \cdot |\sigma_{A=a}T| \geq |S|$ **then**
      5. **For** each $(b, c) \in S$ **do**
         6. **If** $(a, b) \in R$ and $(a, c) \in T$ **then**
            7. Add $(a, b, c)$ to $Q_\Delta$
   8. **else**
      9. **For** each $b \in \pi_B(\sigma_{A=a}R) \land c \in \pi_C(\sigma_{A=a}T)$ **do**
         10. **If** $(b, c) \in S$ **then**
              11. Add $(a, b, c)$ to $Q_\Delta$
12. **Return** $Q$

R and T are in sorted order
Either build indexes, or do a variation of binary search

---

**Algorithm 2: Delaying the Computation**

2.3 Algorithm 2 works as follows. By computing the intersection of two sorted sets takes time at most the power of two choices, computing the intersection of two sorted sets takes time at most the power of two choices, computing the intersection of two sorted sets takes time at most the power of two choices.
Algorithm 2: Delay Computation

For each value \( a_i \), compute valid values of \( B \) that join with it:

\[
\pi_B(\sigma_{A=a_i} R) \cap \pi_B S
\]

For each value of \( b \) in the above result, compute valid values of \( C \):

\[
\pi_C(\sigma_{B=b} S) \cap \pi_C(\sigma_{A=a_i} T).
\]

Can prove to run in : \( O(N) \) on our bad example

General worst-case complexity the same as the previous algorithm
Algorithm 2: Delay Computation

**Algorithm 2** Computing $Q_\Delta$ by delaying computation.

**Input:** $R(A, B), S(B, C), T(A, C)$ in sorted order

1: $Q \leftarrow \emptyset$
2: $L_A \leftarrow \pi_A R \cap \pi_A T$
3: **For** each $a \in L_A$ **do**
4: \hspace{1em} $L_B^a \leftarrow \pi_B \sigma_{A=a} R \cap \pi_B S$
5: \hspace{1em} **For** each $b \in L_B^a$ **do**
6: \hspace{2em} $L_C^{a,b} \leftarrow \pi_C \sigma_{B=b} S \cap \pi_C \sigma_{A=a} T$
7: \hspace{1em} **For** each $c \in L_C^{a,b}$ **do**
8: \hspace{2em} Add $(a, b, c)$ to $Q$
9: **Return** $Q$
Assign weights to each of 
R1, R2, and R3

Say:
R1 → 0.5
R2 → 0.5
R3 → 0.5

Total for B = 0.5 + 0.5 >= 1
B is “covered”
C (1.5), and D (1) are covered
A and E are not covered.

A set of weights is called “fractional edge cover” if all 
attributes are covered
Infinite number of fractional edge covers
Examples, with some fractional edge covers

\[ x_R = \frac{1}{2} \]
\[ x_R = 1 \]
\[ x_S = \frac{1}{2} \]
\[ x_S = 0 \]
\[ x_S + x_T = 1 \]
\[ x_T = \frac{1}{2} \]
\[ x_T = 1 \]
**AGM Bound on Join Sizes**

Why do we care?

Say we have “l” relations in a query \( q \), with sizes \( N_j, j = 1, \ldots, l \)

Let \( u \) denote any fractional edge cover -- so \( u_j \) is the weight for relation with size \( N_j \)

Then, the size of the result is bounded by:

\[
|q| \leq \prod_{j=1}^{l} N_j^{u_j}
\]
AGM Bound on Join Sizes

Using the first cover, result size bounded by:

\[ |Q_\Delta| \leq \sqrt{|R| \cdot |S| \cdot |T|}. \]

If \(|R| = |S| = |T|\), then the bound is \(N^{1.5}\) -- which is tight.

But if \(|R| = |T| = 1\), and \(|S| = N\), then the bound is \(\sqrt{N}\) -- Far from tight -- there can only be 1 triangle.

Using the second cover, result size bounded by:

\[ |Q_\Delta| \leq |R| \cdot |T|. \]

If \(|R| = |S| = |T|\), then the bound is \(N^2\) -- not great.

But if \(|R| = |T| = 1\), and \(|S| = N\), then the bound is 1.
A Generic Algorithm

Algorithm 1: Generic Worst-Case Optimal Join

**given**: A query hypergraph \( H_Q = (V, \mathcal{E}) \) with attributes \( V = \{v_1, \ldots, v_n\} \) and hyperedges \( \mathcal{E} = \{E_1, \ldots, E_m\} \).

**input**: The current attribute index
\( i \in \{1, \ldots, n + 1\} \), and a set of relations \( \mathcal{R} = \{R_1, \ldots, R_m\} \).

```plaintext
function enumerate(i, \mathcal{R})

if \( i \leq n \) then
    // Relations participating in the current join
    \( \mathcal{R}_{\text{join}} \leftarrow \{R_j \in \mathcal{R} \mid v_i \in E_{R_j}\} \);

    // Relations unaffected by the current join
    \( \mathcal{R}_{\text{other}} \leftarrow \{R_j \in \mathcal{R} \mid v_i \notin E_{R_j}\} \);

    // Key values appearing in all joined relations
    foreach \( k_i \in \bigcap_{R_j \in \mathcal{R}_{\text{join}}} \pi_{v_i}(R_j) \) do
        // Select matching tuples
        \( \mathcal{R}_{\text{next}} \leftarrow \{\sigma_{v_i=k_i}(R_j) \mid R_j \in \mathcal{R}_{\text{join}}\} \);

        // Recursively enumerate matching tuples
        enumerate(i + 1, \mathcal{R}_{\text{next}} \cup \mathcal{R}_{\text{other}});

else
    // Produce result tuples
    produce(\( \prod_{R_j \in \mathcal{R}} R_j \));
```

- Process each attribute (variable) at a time
- Find all relations that contain that attribute
- Do an intersection across all the relations for that attribute
- For each value that is present for \( v_i \) in all of \( \mathcal{R}_{\text{join}} \):
  - Select from each relation only those where \( v_i = k_i \)
  - Recurse with those relations plus the rest of the relations
Recap/Thoughts

- Quite a bit of work on this topic in the last 10 years
- Several implementations
  - Often in the context of graph querying
  - Usually require significant pre-computations and specialized indexes
    - The "intersection" step in the previous slide is a key one
  - Some recent work (VLDB 2020) on a more practical implementation using hash indexes instead of sort-based tries
- Still not clear when to use them and when to use binary joins
- Open theoretical issues
- What about outerjoins, etc?
Outline

- Part 1 Slides
  - Query evaluation techniques for large databases, Skew Avoidance, Query compilation/vectorization
  - Query Optimization: Overview, How good are the query optimizers, really?, Reordering for Outerjoins, Query Rewriting

- Adaptive Query Processing

- Worst-case Optimal Join Processing

- Froid: UDFs and Databases
  - Background
  - Froid
User-defined Functions/Procedures

- Supported by database systems since late 80s

```sql
CREATE FUNCTION add(integer, integer) RETURNS integer
AS 'select $1 + $2;'
LANGUAGE SQL
IMMUTABLE
RETURNS NULL ON NULL INPUT;
```

```sql
CREATE OR REPLACE FUNCTION update_influencers_on_insert()
RETURNS TRIGGER
LANGUAGE PLPGSQL
AS $$
declare
  cnt integer;
  username varchar;
BEGIN
  select count(*) into cnt from follows where userid2 = NEW.userid2;
  select max(name) into username from users where userid = NEW.userid2;
  IF cnt = 11 THEN
    insert into influencers values (NEW.userid2, username, cnt);
  ELSEIF cnt > 11 THEN
    update influencers set num_followers = cnt where userid = NEW.userid2;
  END IF;
  RETURN NEW;
END $$;
```
```
User-defined Functions/Procedures

- Supported by database systems since late 80s
- Three main benefits:
  - Modular code
  - Easier to write some code in an imperative language (e.g., ML)
  - Fewer round-trips between application and database
    - Significant performance issues if done repeatedly (e.g., for every order)

```python
c Conn = psycopg2.connect("host=127.0.0.1 dbname=socialnetwork user=postgres password=postgres")
cur = conn.cursor()

cur.execute("drop table if exists influencers;")
cur.execute("create table influencers as select u.userid, u.name, count(user1id) as num_followers from users u join follows f on (u.userid = f.userid) group by u.userid, u.name having count(user1id) > 10;")

cur.execute("drop trigger if exists update_influencers_on_insert on follows;")

cur.execute("drop table if exists friends_small;")
cur.execute("create table friends_small as select f.userid1, f.userid2 from friends f, users u1, users u2 where f.userid1 = u1.userid and f.userid2 = u2.userid and abs(extract(year from u1.birthdate) - extract(year from u2.birthdate)) < 5;")
conn.commit()
```
User-defined Functions/Procedures

- Supported by database systems since late 80s

- Three main benefits:
  - Modular code
  - Easier to write some code in an imperative language (e.g., ML)
  - Fewer round-trips between application and database
    - Significant performance issues if done repeatedly (e.g., for every order)

- Stonebraker notes the latter as the primary reason for adoption of OR features (”what comes around goes around” paper)
  - “Put differently, the major contribution of the OR efforts turned out to be a better mechanism for stored procedures and user-defined access methods.”

- Also called “stored procedures”, with some minor differences across systems
Terminology

- **User-defined functions**
  - Scalar (return a single value) or Table Functions (return a relation)
  - Can be used in queries (WHERE/SELECT/FROM, etc), depending on scalar or table function
  - UDFs typically not allowed to make changes to the database

- **Stored procedures**
  - Similar, but can only be executed using a CALL or EXECUTE command
  - Usually mutate the state of the database

- **Triggers**
  - Something that happens because of an event (e.g., an insert in orders results in an insert in another table)
  - Similar to stored procedures for the actual action
**UDF Challenges**

- **Optimization**
  - UDFs can be very expensive -- coverage() does image analysis of some form
  - Cost of UDFs is hard to estimate -- may depend on the inputs
  - Selectivity of UDFs is hard to estimate -- statistics don’t really help

Example from: “Predicate Migration; Hellerstein and Stonebraker; SIGMOD 1993

```sql
/* Find all maps from week 17 showing more than 1% snow cover. Channel 4 contains images from the frequency range that interests us. */
retrieve (maps.name)
  where maps.week = 17 and maps.channel = 4
  and coverage(maps.picture) > 1
```
UDF Challenges

- **Optimization**
  - UDFs can be very expensive -- `coverage()` does image analysis of some form
  - Cost of UDFs is hard to estimate -- may depend on the inputs
  - Selectivity of UDFs is hard to estimate -- statistics don’t really help

- UDFs cannot be parallelized easily
  - May result in single-threaded execution

- Forces tuple-at-a-time execution
  - Hard to use any of subquery decorrelation techniques

- Often interpreted execution

- Well-known issues resulting in bad performance in many practical scenarios
Outline

- Part 1 Slides
  - Query evaluation techniques for large databases, Skew Avoidance, Query compilation/vectorization
  - Query Optimization: Overview, How good are the query optimizers, really?, Reordering for Outerjoins, Query Rewriting

- Adaptive Query Processing

- Worst-case Optimal Join Processing

- Froid: UDFs and Databases
  - Background
  - Froid
Background on T-SQL

- SQL Server supports: UDFs (cannot modify state), and Stored Procedures (can modify state)

```sql
create function total_price(@key int)
returns char(50) as
begin
    declare @price float, @rate float;
    declare @pref_currency char(3);
    declare @default_currency char(3) = 'USD';
    select @price = sum(o_totalprice) from orders
        where o_custkey = @key;
    select @pref_currency = currency
        from customer_prefs
        where custkey = @key;
    if(@pref_currency <> @default_currency)
        begin
            select @rate =
                xchg_rate(@default_currency, @pref_currency);
            set @price = @price * @rate;
        end
    return str(@price) + @pref_currency;
end
```

```sql
select c_name, dbo.total_price(c_custkey)
from customer;
```

Figure 1: Example T-SQL User defined functions
**UDF Evaluation in SQL Server**

- **Steps**
  - Parsing, binding, normalization: scalar UDFs bound as a UDF operator, but the definition not analyzed
  - Cost-based optimization: Query plans (including for each statement in a UDF) are cached
  - Execution: For each tuple, scalar evaluation sub-system is called
    - May make calls back to the relational execution engine
    - Compilation for an UDF happens on the first call

- **Drawbacks**
  - Iterative invocations (one at a time) -- leads to repeated context switches
  - No costing, Interpreted statement-by-statement (with caching of plans)
  - No intra-query parallelism (as of 2017)
Inline the UDFs by analyzing the code
Froid Framework

- Makes use of APPLY Operator
  - Basically a “flatmap”
  - For each tuple r of R, combine it with each output of E(r) to generate new tuples
    \[ R \ A^\otimes E = \bigcup_{r \in R} (\{ r \} \otimes E(r)) \]
  - The “join” can be: cross product, left outer-join, left-semijion, or left-antijoin

- SQL Server already uses these extensively for subquery decorrelation (as we saw earlier)
Supports imperative constructs in scalar UDFs

**Table 1:** Relational algebraic expressions for imperative statements (using standard T-SQL notation from [33])

<table>
<thead>
<tr>
<th>Imperative Statement (T-SQL)</th>
<th>Relational expression (T-SQL)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DECLARE {@var data_type [= expr]}[,...n];</td>
<td>SELECT {expr</td>
</tr>
<tr>
<td>SET {@var = expr}[,...n];</td>
<td>SELECT {expr AS var}[,...n];</td>
</tr>
<tr>
<td>SELECT {@var1 = prj_expr1}[,...n] FROM sql_expr;</td>
<td>{SELECT prj_expr1 AS var1 FROM sql_expr}; [,...n]</td>
</tr>
<tr>
<td>IF (pred_expr) {t_stmt;[,...n]} ELSE {f_stmt;[,...n]}</td>
<td>SELECT CASE WHEN pred_expr THEN 1 ELSE 0 END AS pred_val;</td>
</tr>
<tr>
<td>RETURN expr;</td>
<td>{SELECT CASE WHEN pred_val = 1 THEN t_stmt ELSE f_stmt; }[,...n]</td>
</tr>
<tr>
<td></td>
<td>SELECT expr AS returnVal;</td>
</tr>
</tbody>
</table>
UDF Algebrization

- Construction of regions
  - Basic sequential regions, condition regions (if-else), and loop regions (loops)
  - Hierarchical (regions can contain regions)
- Relational expressions for each region
  - Variable declarations/assignments

```
set @default_currency = 'USD';
select @price = sum(o_totalprice) from orders
  where o_custkey = @key;

select (select sum(o_totalprice) from orders
  where o_custkey = @key) as price
```
UDF Algebrization

- Relational expressions for each region
  - Variable declarations/assignments
  - Conditional statements

```sql
if (@total > 1000)
    set @val = 'high';
else
    set @val = 'low';
```

- Return statements
  - Code may have multiple return points
  - Modeled as a “jump” to the end of the codeblock
  - Implemented through use of “probe” and “pass-through” of APPLY
UDF Algebrization

- Combining expressions for multiple statements
  - For each statement: compute a “read-set” and a “write-set”

Table 2: Derived tables for regions in function total_price.

<table>
<thead>
<tr>
<th>Region</th>
<th>Write-sets (Derived table schema)</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>DT1 (price float, rate float, default_currency char(3), pref_currency char(3))</td>
</tr>
<tr>
<td>R2</td>
<td>DT2 (price float, rate float)</td>
</tr>
<tr>
<td>R3</td>
<td>DT3 (returnVal char(50))</td>
</tr>
</tbody>
</table>

Use these as the “schemas” of derived tables to be computed.
Combining expressions for multiple statements

- For each statement: compute a “read-set” and a “write-set”
- Use these as schemas of derived tables
- Connect the regions using APPLY (with pass-through in case of multiple return statements)

Correctness?

- Each individual transformation correct by itself
- All derived tables contain a single tuple
- Outer apply preserves the semantics of combined execution

Note: Doesn’t handle loops -- may be trickier to model
Replace the scalar UDF with the relational expression (not as SQL, but rather operators)

Let the optimizer de-correlate and optimize

Resulting plan looks complex, but decorrelates as desired

Figure 5: Plan for inlined UDF total_price of Figure 1
Dynamic slicing: use compile-time constants to simplify queries

Constant folding and propagation: already done by SQL server

Dead code elimination: optimizer handles these during project pushdown

```
create function getVal(@x int) 
returns char(10) as
begin
  declare @val char(10);
  if(@x > 1000)
    set @val = 'high';
  else set @val = 'low';
  return @val + ' value';
end
```

(a) Input UDF

```
(i) Dynamic slicing for getVal(5000)
begin
  declare @val char(10);
  set @val = 'high';
  return 'high value';
end

(ii) Constant propagation & folding
begin
  declare @val char(10);
  set @val = 'high';
  return 'high value';
end

(iii) Dead code elimination
begin
  return 'high value';
end
```

(b) Common optimizations done by an imperative language compiler

```
select returnVal from
(select case when @x > 1000
then 'high' else 'low' end as val) DT1
outer apply
(select DT1.val + ' value' as returnVal) DT2
```

(c) Output of FROID's Algebrization

```
select returnVal from
(select 'high' as val) DT1
outer apply
(select DT1.val + ' value' as returnVal) DT2
```

(d) How FROID achieves the same end result as Figure 5(b) using relational algebraic transformations

**Figure 5:** Compiler optimizations as relational transformations. For ease of presentation, (c) and (d) are shown in SQL; these are actually transformations on the relational query tree representation.
Design and Implementation

- Should this inlining be done in a cost-based manner?
  - Influences whether it takes place during binding or during query optimization
  - Experiments showed it is almost always beneficial + hard to modify optimizers ➔ do it in the binding phase

- Constraints
  - Put a constraint on the maximum size of UDFs that can be algebrized

- Froid is extensible -- could handle other languages as well

- Security and permissions
  - A user may not have permission on the UDF but on the tables, and vice versa
  - Need to be careful with caches as well
Evaluation

- Applicability
  - Used top 100 customer workloads from Azure SQL → 85329 scalar UDFs
  - Froid could handle 60% or so

```sql
create function dbo.F1(@p1 int, @p2 int)
returns bit as
begin
  if EXISTS
    (SELECT 1 FROM View1 WHERE col1 = 0
     AND col2 = @p1
     AND ((col2 = 2) OR (col3 = 2))
     AND dbo.F2(col4, @p2, 0) = 1
     AND dbo.F2(col5, @p2, 0) = 1
     AND dbo.F2(col6, @p2, 0) = 1
     AND dbo.F2(col7, @p2, 0) = 1
     AND dbo.F2(col8, @p2, 0) = 1
     AND dbo.F2(col9, @p2, 0) = 1
     AND dbo.F2(col10, @p2, 0) = 1
     AND dbo.F2(col11, @p2, 0) = 1
     AND dbo.F2(col12, @p2, 0) = 1
     AND dbo.F2(col13, @p2, 0) = 1
     AND dbo.F2(col14, @p2, 0) = 1
     AND dbo.F2(col15, @p2, 0) = 1)
  return 1
  return 0
end
```

```sql
create function dbo.VersionAsFloat(@v nvarchar(96))
returns float as
begin
  if @v is null return null
  declare @first int, @second int;
  declare @major nvarchar(6), @minor nvarchar(10);

  set @first = charindex('.', @v, 0);
  if @first = 0
    return CONVERT(float, @v);

  set @major = SUBSTRING(@v, 0, @first);
  set @second = charindex('.', @v, @first + 1);
  if @second = 0
    set @minor = SUBSTRING(@v, @first + 1, len(@v) - @first)
  else
    set @minor = SUBSTRING(@v, @first + 1, @second - @first - 1);

  set @minor = CAST(CAST(@minor AS int) AS varchar);
  return CONVERT(float, @major + '.' + @minor);
end
```

```sql
CREATE FUNCTION dbo.RptBracket(@MyDiff int, @NDays int)
RETURNS nvarchar(10) AS
BEGIN
  if(@MyDiff >= 5 * @NDays)
  begin
    RETURN ( Cast(5 * @NDays as nvarchar(5)) + N'+' )
  end
  RETURN ( Cast(Floor(@MyDiff / @NDays) * @NDays as nvarchar(5))
    + N' - '
    + Cast(Floor(@MyDiff / @NDays + 1) * @NDays - 1 as nvarchar(5)) )
END
```
**Evaluation**

![Graph 6](image.png)

**Figure 6:** Varying the number of UDF invocations

![Graph 10](image.png)

**Figure 10:** Improvement for UDFs in workload W1
Converting UDFs to CTEs

- CTE == Common Table Expressions (i.e., WITH clause)
- Another approach taken by a recent paper
  - Functional-style SQL UDFs with a Capital ‘F’; SIGMOD 2020

```
CREATE FUNCTION pow(x int, n int)
RETURNS int AS $$
DECLARE
  i int = 0;
  p int = 1;
BEGIN
  WHILE i < n LOOP
    p = p * x;
    i = i + 1;
  END LOOP;
RETURN p;
END;
$$
```

```
WITH RECURSIVE
  run("call?",i1,p1,x,n,result) AS (
    SELECT true,0,1,x,n,NULL
    UNION ALL
    SELECT iter.* FROM run, LATERAL (SELECT false,0,0,0,0,p1 WHERE i1 >= n
    UNION ALL
    SELECT true,i1+1,p1*x,x,n,0 WHERE i1 < n
  ) AS iter("call?",i1,p1,x,n,result)
    WHERE run."call?"
) AS iter("call?",i1,p1,x,n,result)
SELECT * FROM run;
```