DSE 203

DAY 4: VIRTUAL DATA INTEGRATION
Recap: Architecture for Virtual Data Integration
Mediators

• A mediator is a software system that offers a common query interface to a set of heterogeneous information sources

• Components of a mediator-based data integration system
  • Collection of data sources that are independent and autonomous
  • A virtual “database” is created which is accessed via the mediator through wrappers
    • Wrappers
      • Translates mediator’s query to queries/functions that a data source can process
      • Translates the data source’s output into a data object the mediator can process
  • A mediator that creates the illusion on users of being interacting with a real database
  • Queries are posed and answered via the mediator
Tasks of a Mediator

- **Accept the participation** of different data sources
- **Contains information about the contents of the data sources**
- **Collects data** from sources upon request *at query time*
- **Logically integrates** the different data sources by means of a unifying, global or mediated schema
- **Receives queries from users** that are expressed in the language of the global schema
- In order to answer global queries, it **sends appropriate queries to the sources**
- **Combines the answers** received from the sources to construct the final answer for the user query
Issues to Remember for Virtual Data Integration - I

• Sources are autonomous
  • They operate independently of each other (mostly)
    • Independent content
    • Independent updates
    • Independent availability, e.g., due to participation in other data systems
  • The mediator cannot update the data content or change the query behavior of any source
  • Sources may join or leave the system at will

• Source Descriptions are available to the mediator
  • Schemas if available
  • Match and mapping specified
Issues to Remember for Virtual Data Integration - II

- The mediator should allow all sources to participate
  - Number of active sources can vary
- Each application will create its own global schema
- The system is responsible for solving/addressing problems with data
  - *Redundancy*: to avoid unnecessary computations
  - *Complementarity*: data of the same kind may be spread through different sources and has to be detected and combined
  - *Consistency*: two sources, independently, may be consistent, but taken together, possibly not
    - Same ID card number may be assigned to different people in different sources
Parts of a Mediator
Query Reformulation

• Problem: rewrite a user query expressed in the mediated schema into a query expressed in the source schema

Given a query $Q$ in terms of the mediator schema relations, and descriptions of information sources

Find a query $Q'$ that uses only the source relations, such that

• $Q' \subseteq Q$, and
• $Q'$ provides all possible answers to $Q$ given the sources
Query Rewriting Using Views

• **Query Containment**: $q' \subseteq q \iff \forall D \ q'(D) \subseteq q(D)$

• **Query Equivalence**: $q' \equiv q \iff q' \subseteq q \land q \subseteq q'$

Given query $q$ and view definitions $V = \{v_1, \ldots, v_n\}$

• $q'$ is an **Equivalent Rewriting** of $q$ using $V$ if
  • $q'$ refers only to views in $V$, and
  • $q' \equiv q$

• $q'$ is an **Maximally-Contained Rewriting** of $q$ using $V$ if
  • $q'$ refers only to views in $V$ and
  • $q' \subseteq q$, and
  • There is no rewriting $q''$, such that $q' \subseteq q''$ and $q'' \not\equiv q'$
Relationship between Local Schema and Global Schema

- Defining views to specify the schema mappings between the global and local schemata
- Global as View (GAV)
- Local as View (LAV)
- Global-Local as View (GLAV)
- We only consider the first two
Global as View

• Each relation in the mediator’s schema is written as a view over some combination of the source schemata

• Example
  • Local Sources
    • DB1(Title, Dir, Year )
    • DB2(Title, Dir, Year )
    • DB3(Title, Review)
  • Mediator’s Schema (RM1)
    • MovieYear (Title, Year ) ← DB1(Title, Dir, Year )
    • MovieYear (Title, Year ) ← DB2(Title, Dir, Year )
  • Algebra Expression
    • MovieYear := Π\text{Movie, Year}(DB_1) \cup Π\text{Movie, Year}(DB_2)

• Mediator’s Schema (RM2)
  • MovieRev(Title, Dir, Review) ← DB1(Title, Dir, Year), DB3(Title, Review)
  • Algebra Expression
    • MovieRev := Π_{\text{Title, Dir, Review}}(DB_1 \bowtie \text{Title} DB_3)

• User’s Query: Movies shown in year 2010, with their reviews
  • Ans(Title, Review) ← MovieYear (Title, 2010), MovieRev(Title, Dir, Review)
Query Answering in GAV

• Query
  • \textit{Ans}(\textit{Title}, \textit{Review}) \leftarrow \text{MovieYear}(\textit{Title}, 2010), \text{MovieRev}(\textit{Title}, \textit{Dir}, \textit{Review})

• Query Rewriting by view unfolding (also called rule unfolding)
  • \textit{Ans}(\textit{Title}, \textit{Review}) \leftarrow \text{DB}_1(\textit{Title}, \textit{Dir}, 2001), \text{DB}_1(\textit{Title}, \textit{Dir}, 2001), \text{DB}_3(\textit{Title}, \textit{Review})
  • \textit{Ans}(\textit{Title}, \textit{Review}) \leftarrow \text{DB}_2(\textit{Title}, \textit{Dir}, 2001), \text{DB}_1(\textit{Title}, \textit{Dir}, 2001), \text{DB}_3(\textit{Title}, \textit{Review})

• Removing redundant subgoal in body
  • \textit{Ans}_1(\textit{Title}, \textit{Review}) \leftarrow \text{DB}_1(\textit{Title}, \textit{Dir}, 2001), \text{DB}_3(\textit{Title}, \textit{Review})

• Query Containment
  • \textit{Ans}_2(\textit{Title}, \textit{Review}) \subseteq \textit{Ans}_1(\textit{Title}, \textit{Review})

What does this imply?
Wrinkle 1: Access-Pattern Restrictions

- Every source has a **wrapper**
  - Some sources require certain access patterns or input bindings
    - we represent with adornments, e.g., CitationDB^{bf}(X,Y)

- The optimizer must find a plan that joins CitationDB’s X attribute with another relation expression
  - ... for which we don’t have binding restrictions!
  - Requires a **dependent join** operation, which is implemented like the 2-way **semijoin** but feeds data to the wrapper
Plan with a Dependent Join

\[ \pi_{title,startTime} \quad \text{DepJoin}_{title = movie} \]

Feeds data from left input to the wrapper in the right input.
Wrinkle 2: Source Query Capabilities

• Some sources can execute certain operations (selection, project, sometimes join or more)
  e.g., an SQL database as a source
  ... or even a Web form that applies filter predicates

• The data integration system optimizer must be able to estimate the cost of pushing operations into the source
  • ... which in turn may vary depending on the source’s load, which we may not know...
Plan with a Dependent Join

Feeds data from left input to the wrapper in the right input

When can we do this?
The Basic Approach

Enumerate all possible query expressions

- “bushy” plan enumeration instead of “left-linear” as done for relational DBMS optimization

Use a different set of cost formulas for pushed-down operations, custom to the wrapper and the source

- left-linear plan
- non-linear (“bushy”) plan

joins with single relations on right

joins of arbitrary expressions
Local As View

• The mediator’s schema is known
• Every source maps its schema as a view over the mediator’s schema
• The query is placed against the mediator’s schema and needs to be transformed by “inverting” the mapping to construct queries against the sources
LAV Example

• Mediator’s Schema
  • $Movie(Title, Year, Director, Genre)$
  • $AmerDir(Director)$
  • $Review(Title, Review)$

• $S_1$: Has a relation $V_1(Title, Year, Director)$ containing comedies, filmed after 1960, with American directors and their years
  • In terms of the mediator
    • $V.(Title, Year, Director) ← Movie(Title, Year, Director, Genre), AmerDir(Director), Genre = comedy, Year ≥ 1960.$

• $S_2$: Has a relation $V_2(Title, Review)$ containing movies filmed after 1990 with their reviews, but no directors
  • In terms of the mediator
    • $V.(Title, Review) ← Movie(Title, Year, Director, Genre), Review(Title, Review), Year ≥ 1990.$
Querying Mediators with LAV Mappings

• Find Comedies with their reviews produced since 1950
  • \( \text{Ans}(\text{Title,Review}) \leftarrow \text{Movie}(\text{Title, Year, Director, comedy}), \text{Review}(\text{Title,Review}), \text{Year} \geq 1950. \)

• Not possible to obtain answers by a simple, obvious or direct computation of the RHS of the query

• No simple rule unfolding for the relations in the body: no definitions for them as in GAV

\[ \text{V}(\text{Title, Year, Director}) \leftarrow \text{Movie}(\text{Title, Year, Director, Genre}), \text{AmerDir}(\text{Director}), \text{Genre} = \text{comedy}, \text{Year} \geq 1960. \]
\[ \text{V}(\text{Title, Review}) \leftarrow \text{Movie}(\text{Title, Year, Director, Genre}), \text{Review}(\text{Title, Review}), \text{Year} \geq 1990. \]
A Good Rewriting (with Some Caveats)

• Query
  • $\text{Ans}(\text{Title, Review}) \leftarrow \text{Movie}(\text{Title, Year, Director, comedy}), \text{Review}(\text{Title, Review}), \text{Year} \geq 1950$.

• Rewriting
  • $\text{Ans}(\text{Title, Review}) \leftarrow (V_1(\text{Title, Year, Director}), \text{Year} \geq 1950), (V_2(\text{Title, Review}))$

• Query is rewritten in terms of the views; and can be computed:
  • Extract values for $\text{Title}$ from $V_1$
  • Extract the tuples from $V_2$
  • At the mediator level, compute the join via $\text{Title}$
  • Note: The results are only partial – why?
Which of These Views Can be Used to Answer Q?

\[ Q(T,Y,D) : \neg \text{Movie}(I,T,Y,G), Y \geq 1950, G = "comedy" \]

\[ \text{Director}(I,D), \text{Actor}(I,D) \]

\[ V_2(I,T,Y) : \neg \text{Movie}(I,T,Y,G), Y \geq 1950, G = "comedy" \]

\[ V_3(I,D) : \neg \text{Director}(I,D), \text{Actor}(I,D) \]

\[ V_6(T,Y) : \neg \text{Movie}(I,T,Y,G), Y \geq 1950, G = "comedy" \]

\[ V_7(I,T,Y) : \neg \text{Movie}(I,T,Y,G), Y \geq 1950, \]
\[ G = "comedy", \text{Award}(I,W) \]

\[ V_8(I,T) : \neg \text{Movie}(I,T,Y,G), Y \geq 1940, G = "comedy" \]
The Bucket Algorithm

Key idea:

- Create a bucket for each subgoal $g$ in the query.
- The bucket contains view atoms that contribute to $g$.
- Create rewritings from the Cartesian product of the buckets.
Bucket Algorithm in Action

\[ Q(ID,Dir) : \neg \text{Movie}(ID,\text{title},\text{year},\text{genre}), \text{Revenues}(ID,\text{amount}), \]
\[ \text{Director}(ID,\text{dir}), \text{amount} \geq 100M \]
\[ V_1(I,Y) : \neg \text{Movie}(I,T,Y,G), \text{Revenues}(I,A), I \geq 5000, A \geq 200M \]
\[ V_2(I,A) : \neg \text{Movie}(I,T,Y,G), \text{Revenues}(I,A) \]
\[ V_3(I,A) : \neg \text{Revenues}(I,A), A \leq 50M \]
\[ V_4(I,D,Y) : \neg \text{Movie}(I,T,Y,G), \text{Director}(I,D), I \leq 3000 \]

View atoms that can contribute to Movie:
\[ V_1(ID,\text{year}), V_2(ID,A), V_4(ID,D',\text{year}) \]
The Buckets and Cartesian product

<table>
<thead>
<tr>
<th>Movie(ID,title, year,genre)</th>
<th>Revenues(ID, amount)</th>
<th>Director(ID,dir)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$V_1(ID,year)$</td>
<td>$V_1(ID,Y')$</td>
<td>$V_4(ID,Dir,Y')$</td>
</tr>
<tr>
<td>$V_2(ID,A')$</td>
<td>$V_2(ID,amount)$</td>
<td></td>
</tr>
<tr>
<td>$V_4(ID,D',year)$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Consider first candidate rewriting: first $V_1$ subgoal is redundant, and $V_1$ and $V_4$ are mutually exclusive.

$$q_1'(ID,dir) : -V_1(ID,year), V_1(ID,y'), V_4(ID,dir,y')$$
Next Candidate Rewriting

<table>
<thead>
<tr>
<th>Movie(ID,title,year,genre)</th>
<th>Revenues(ID,amount)</th>
<th>Director(ID,dir)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$V_1$(ID,year)</td>
<td>$V_1$(ID,Y')</td>
<td>$V_4$(ID,Dir,Y')</td>
</tr>
<tr>
<td>$V_2$(ID,A')</td>
<td>$V_2$(ID,amount)</td>
<td></td>
</tr>
<tr>
<td>$V_4$(ID,D’,year)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$q_2'(ID,dir) : -V_2(ID,A'), V_2(ID,amount), V_4(ID,dir,y')$
$q_2'(ID,dir) : -V_2(ID,amount), V_4(ID,dir,y'), amount \geq \$100M$
The Bucket Algorithm: Summary

• Cuts down the number of rewriting that need to be considered, especially if views apply many interpreted predicates.

• The search space can still be large because the algorithm does not consider the interactions between different subgoals.
  • See next example.
The MiniCon Algorithm

\[ Q(title, year, dir) : \neg Movie(ID, title, year, genre), \]
\[ \quad Director(ID, dir), \ Actor(ID, dir) \]
\[ V_5(D, A) : \neg Director(I, D), \ Actor(I, A) \]

Intuition: The variable \( I \) is not in the head of \( V_5 \), hence \( V_5 \) cannot be used in a rewriting. MiniCon discards this option early on, while the Bucket algorithm does not notice the interaction.
MinCon Algorithm Steps

• Create MiniCon descriptions (MCDs):
  • Homomorphism on view heads
  • Each MCD covers a set of subgoals in the query with a set of subgoals in a view

• Combination step:
  • Any set of MCDs that covers the query subgoals (without overlap) is a rewriting
  • No need for an additional containment check!
MiniCon Descriptions (MCDs)
An atomic fragment of the ultimate containment mapping

\[ Q(title,act,dir) : \neg Movie(ID,title,year,genre), \]
\[ \quad Director(ID,dir), Actor(ID,act) \]

\[ V(I,D,A) : \neg Director(I,D), Actor(I,A) \]

MCD:

<table>
<thead>
<tr>
<th>mapping:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ID \rightarrow I</td>
<td></td>
</tr>
<tr>
<td>dir \rightarrow D</td>
<td></td>
</tr>
<tr>
<td>act \rightarrow A</td>
<td></td>
</tr>
</tbody>
</table>

covered subgoals of Q: \{2,3\}
MCDs: Detail 1

\[ Q(title, year, dir) : \neg Movie(ID, title, year, genre), \]
\[ \quad Director(ID, dir), Actor(ID, dir) \]

\[ V(I,D,A) : \neg Director(I,D), Actor(I,A) \]

Need to specialize the view first:

\[ V'(I,D,D) : \neg Director(I,D), Actor(I,D) \]

MCD:

<table>
<thead>
<tr>
<th>ID</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>dir</td>
<td>D</td>
</tr>
</tbody>
</table>

mapping:

covered subgoals of Q: \{2,3\}
MCDs: Detail 2

\[ Q(\text{title}, \text{year}, \text{dir}) : \neg \text{Movie}(\text{ID}, \text{title}, \text{year}, \text{genre}), \]
\[ \text{Director}(\text{ID}, \text{dir}), \text{Actor}(\text{ID}, \text{dir}) \]
\[ V(\text{I}, \text{D}, \text{D}) : \neg \text{Director}(\text{I}, \text{D}), \text{Actor}(\text{I}, \text{D}), \]
\[ \text{Movie}(\text{I}, \text{T}, \text{Y}, \text{G}) \]

Note: the third subgoal of the view is *not* included in the MCD.

**MCD:**

- **Mapping:**
  - \( ID \rightarrow I \)
  - \( dir \rightarrow D \)

- **Covered subgoals of \( Q \) still:** \{2,3\}
DSE 203

DAY 4: QUERY PROCESSING IN A VIRTUAL INTEGRATION SYSTEM
Query Processing in Virtual Integration Systems

- Internet-based sources present challenges
  - Slow and unpredictable data transfer rates
  - Source failures

- Many data integration applications also emphasize early results!

- Emphasize maximally pipelined query processing + failure handling...
Threads to the Rescue

• Different sources will send data at different rates – an architecture based on iterators may hold up the CPU when it’s waiting for I/O
  • e.g., we are trying to read one table but no data is yet ready

• Develop fully pipelined join operators with multiple threads (one to process each input, one for the computation)...
  • Example: pipelined hash join
Pipelined Hash Join
(aka Symmetric Hash Join)

Operates symmetrically, and is fully pipelined:

1. Read from either input
2. Add tuple to input’s hash table
3. Probe against opposite hash table
4. Return resulting output
Pipelined Hash Join

Each red ellipse represents a separate thread of control, scheduled by the CPU based on availability.
Pipelined Join Plan
Other Pipelined Operators

• In general, all algorithms will be hash-based as opposed to index- or sort-based
  This supports pipelined execution in a networked setting

• Sometimes this requires techniques for when hash tables exceed memory capacity
  • Spill to disk, process by recursively applying hashing or by using nested loops algorithms
A Problem: Problems Can Arise!

- It’s not uncommon for a data source to become unavailable
  - The server crashes, the network becomes unavailable, a job gets preempted, etc.

- Operators in a data integration query plan need to generate *events*, much like exceptions in a programming environment
  - These need to be handled by the query processor
Adaptive Query Processing

Distributed, pipelined execution is a necessity to handle data integration queries

However, we can still end up with plans that:
• Encounter massive delays for some sources
• Run into errors – e.g., a source fails
• Are highly inefficient – e.g., produce huge amounts of intermediate state due to bad cost estimates

This motivates adaptive query processing
The First Step: A Unified Query Processor

- Most relational DB engines have separate query optimization and execution modules
  - (This is in fact changing)
- Our goal here: start executing a query and adapt to conditions
- Requires a unified query processor that loops between decision-making and execution
Adaptive Query Processing in Data Integration

[Diagram showing stages of adaptive query processing: Parsing, Query reformulation, System catalog, Source descriptions, Query (re)optimization, Query execution, and Data sources. Each step involves operations such as parsing, reformulation, statistics & costs, and incremental query plan.]
Challenges of Adaptive Query Processing

• Generally a balance among many factors:
  • How much information to collect – exploration vs. exploitation
  • Analysis time vs. execution time – more optimization results in better plans, but less time to spend running them
  • Space of adaptations – how much can we change execution?
  • Short-term vs. long-term – balancing between making the best progress **now**, and incurring state that might cost a lot **later**.

• We divide into **event**- and **performance-driven** types
Event-Driven Adaptivity

Consider changing the plan at designated points where the query processor is stopped

• When a source fails or times out
• When a segment of the query plan complete and is materialized
  • ... and we verify at runtime that the materialized result size is significantly different from expectations

➢ We can encode decision-making for the execution system through a rule mechanism – on event if condition then take action... including re-running the optimizer!
Source Failure Events and Rules

• Source failures can occur in two main ways:
  • a source times out
  • a source is unreachable

    \texttt{on \ timeout(wrapper1, 10msec) ...}

• There can be multiple kinds of responses
  • Find an alternative source
    \texttt{on \ timeout(wrapper1, 10msec) \ \textbf{if} \ \text{true} \ \textbf{then} \ \text{activate}(coll1, A)}
  • Reschedule (suspend until later) a subplan or operator
    \texttt{on \ timeout(wrapper1, 10msec) \ \textbf{if} \ \text{true} \ \textbf{then} \ \text{reschedule}(op2)}
How Do We Generate the Rules?

• For source failures, we need a partial ordering of data sources with alternates
  • The optimizer can generate rules to access the alternates if the primaries are down

• For rescheduling, the optimizer can defer parts of the plan that are dependent on a blocked data source
Cost Mis-estimation Events

• Another source of problems: the optimizer mis-estimates the cost or cardinality of a result
  • ... and in turn this may cause sub-optimal execution of the rest of the plan!

• Can we find points at which to change the rest of the plan, based on what we discover in early execution?
  • Gather information during execution – special operators to track cardinalities, build histograms
  • Periodically “checkpoint” and consider re-optimizing
  • If necessary, re-optimize with the results we’ve computed
Mid-query Re-optimization

- Break the plan into several pipelines, e.g., based on uncertainty of results
  - Write the results to disk, place a “checkpoint”

- During each stage, add statistics collection operations
  - e.g., histograms on join attributes

- At the checkpoint, re-estimate costs
  - If they are beyond a threshold, re-run the optimizer
Re-Optimization

• In an event-driven setting, the optimizer only gets re-run if it looks likely that costs for the remaining plan will be significantly different

• The existing result is typically treated as a **materialized view** and the initial query can be re-optimized
  • If useful, the materialized view will be directly used by the optimizer
  • In the worst case, the size of the view, plus the statistics gathered, result in a better cost estimate
Challenges of Event-Driven Adaptivity

• Checkpoints are expensive – they prevent fully pipelined execution

• It is hard to decide where to put the checkpoint!

• Sometimes we will waste a lot of work in getting to the checkpoint

• This motivates performance-driven adaptivity...
Performance-Driven Adaptivity

• Suppose we want to change query plans in mid-execution, without re-reading data

• Let’s exploit the fact that multiple relational expressions can do equivalent computation
  • Break up the data into segments or phases
  • Run a different plan over each phase
  • Put the results together!

• By the relational algebra:

$$\Pi_{A}(\sigma(R_1 \bowtie \ldots \bowtie R_m)) \equiv \bigcup \Pi_{A}(\sigma(R_{c_1}^{i} \bowtie \ldots \bowtie R_{c_m}^{i})))_{1 \leq c_1 \leq n, \ldots, 1 \leq c_m \leq n}$$
Corrective Query Processing

• The basic idea of **corrective query processing:**
  • Frequent optimize-execute loop
  • We monitor progress and continuously re-estimate cost
  • If cost exceeds some threshold, we **suspend** the current query plan, use the optimizer to choose a new plan

  • The new plan “resumes” from the input streams that the old plan was using
  • Ultimately we need to combine data across these plans (phases) – this is what we call the **stitch-up phase**
Corrective Query Processing
How Cost Re-Estimation and Re-optimization Work

• Every few seconds, plan status is polled
  • Cardinalities are re-estimated based on “progress so far” (based on an estimate of how much more data remains)

• The query optimizer’s cost estimator is re-run over the current plan*
  • If it diverges by a threshold, re-optimization is triggered in the background (while execution continues)
  • Re-optimization uses updated selectivity + cost values

• If a “significantly better” plan is found, the current plan is suspended and the new one is started

* The optimizer remains in memory at all times
Stitch-up Plans

• A challenge: “new” results need to be joined with “old ones”

• We can do this in three ways:
  • Feed new results into old plans, join with old data
  • Feed old results into new plans, join new data with old
  • Do all of the cross-plan joining in a separate stage (phase)

• The last one is called a stitch-up plan and can be done with special join algorithms that efficiently compute only those answers that remain
Query Processing Wrap-up

• Query processing for data integration builds upon past models
  • From centralized DBMSs: plan enumeration, cost modeling, basic query execution architectures
  • From distributed DBMSs: modeling communication cost, data distribution algorithms

• But a new emphasis on:
  • Wrappers, access patterns, and query subexpression push-down
  • Fully pipelined algorithms, multithreaded operation
  • Adaptivity – event-based and performance-based
DSE 203

DAY 4: CROSS-MODEL DATA INTEGRATION
Model Transformation

• Scenario
  • Source and mediator have different data models
  • Data transformation from source model to mediator model
  • Query transformation from mediator’s query language to the source’s query language

• We will look at some examples
Case 1: XML and Relational

• Many solutions
  • Convert relational data to EER representation $\rightarrow$ convert EER to XML schema $\rightarrow$ mediator queries XML schema
  • Create relational view of XML data $\rightarrow$ transform XML query to relational query against the view $\leftarrow$ write source's relational schema to the XML view
  • Create an XQuery to SQL converter $\rightarrow$ package the results into an XML document
  • Create XML view of relational data $\rightarrow$ rewrite XML queries to queries against the views $\rightarrow$ transform these queries into SQL $\rightarrow$ compose results into XML
  • ...

XPERANTO (IBM Almaden)

- A simple purchase order database – relational and its default XML view

```
<db>
  <order>
    <row id="10"><custname>Smith Construction</custname> <custnum>7734</custnum></row>
    <row id="10"><custname>Western Builders</custname> <custnum>7725</custnum></row>
  </order>
  <item>
    <row oid="10"><desc>generator</desc> <cost>8000</cost></row>
    <row oid="10"><desc>backhoe</desc> <cost>24000</cost></row>
  </item>
  <payment>
    ... similar to <order> and <item>
  </payment>
</db>
```
XML Views

• The desired view (purchase order)

```xml
<order id="10">
  <customer> Smith Construction </customer>
  <items>
    <item description="generator"> <cost> 8000 </cost> </item>
    <item description="backhoe"> <cost> 24000 </cost> </item>
  </items>
  <payments>
    <payment due="1/10/01"> <amount> 20000 </amount> </payment>
    <payment due="6/10/01"> <amount> 12000 </amount> </payment>
  </payments>
</order>
```

• The view definition

```sql
01. create view orders as (  
02.   for Order in view("default")/order/row  
03.   return  
04.   <order id=$order/id>  
05.   <customer> $order/custname </customer>  
06.   <items>  
07.     for $item in view("default")/item/row  
08.     where $order/id = $item/oid  
09.     return  
10.     <item description=$item/desc >  
11.       <cost> $item/cost </cost>  
12.     </item>  
13.   </items>  
14.   <payments>  
15.     for $payment in view("default")/item/row  
16.     where $order/id = $payment/oid  
17.     return  
18.     <payment due=$payment/date>  
19.       <amount> $payment/amount </amount>  
20.     </payment>  
21.     sortby(@due)  
22.   </payments>  
23. )
```
Query Graph for the view

- A node is an operation on a data structure containing the result of the previous operation.
- The output of the operation is a tuple as marked on top of the node.
- An edge represents the flow of intermediate results from one operator to the next.
Querying the XML view

1. for $order in view("orders")
2. where $order/customer/text() like "Smith%"
3. return $order

- Correlated join
  - Join whose inputs are correlated queries

- Correlated subqueries
  - Queries where variables of an outer query are shared by the inner subquery

- Query decorrelation
  - Rewriting the query such that the inner query doesn’t get executed for every value of the correlating variable of outer query
Composing the query with the view

correlation on order.id

groupby: $items = aggXMLFags($item)

project: $item = <item> ...

select: $oid = $id

table: item

join (correlated):

Sorder Scustname Sitems Spoints

project: Sorder = <order> ...

groupby: $points = aggXMLFags($point)

orderby (on $due): $points = <payment> ...

select: $id = $oid

table: order

Soid Sdesc Scost

table: item

Sid Scustname

table: order

Scustname like 'Smith%'

Sorder join (correlated):

correlation on Scustname

select: Scustname like 'Smith%'

table: order

Predicate pushdown

Composing the query with the view
Case 2: Relational and Graph

User (US)

<table>
<thead>
<tr>
<th>uid</th>
<th>uname</th>
</tr>
</thead>
<tbody>
<tr>
<td>t₁</td>
<td>t₂</td>
</tr>
<tr>
<td>u₀₁</td>
<td>Date</td>
</tr>
<tr>
<td>u₀₂</td>
<td>Hunt</td>
</tr>
</tbody>
</table>

Follower (FR)

<table>
<thead>
<tr>
<th>fuser</th>
<th>fblog</th>
</tr>
</thead>
<tbody>
<tr>
<td>t₃</td>
<td>u₀₁</td>
</tr>
<tr>
<td></td>
<td>b₀₁</td>
</tr>
<tr>
<td>t₄</td>
<td>u₀₁</td>
</tr>
<tr>
<td></td>
<td>b₀₂</td>
</tr>
<tr>
<td>t₅</td>
<td>u₀₁</td>
</tr>
<tr>
<td></td>
<td>b₀₃</td>
</tr>
<tr>
<td>t₆</td>
<td>u₀₂</td>
</tr>
<tr>
<td></td>
<td>b₀₁</td>
</tr>
</tbody>
</table>

Tag (TG)

<table>
<thead>
<tr>
<th>t₇</th>
</tr>
</thead>
<tbody>
<tr>
<td>tuser</td>
</tr>
<tr>
<td>tcomment</td>
</tr>
<tr>
<td>u₀₂</td>
</tr>
<tr>
<td>c₀₁</td>
</tr>
</tbody>
</table>

Blog (BG)

<table>
<thead>
<tr>
<th>bid</th>
<th>bname</th>
<th>admin</th>
</tr>
</thead>
<tbody>
<tr>
<td>t₈</td>
<td>Information Systems</td>
<td>u₀₂</td>
</tr>
<tr>
<td>t₉</td>
<td>Database</td>
<td>u₀₁</td>
</tr>
<tr>
<td>t₁₀</td>
<td>Computer Science</td>
<td>u₀₂</td>
</tr>
</tbody>
</table>

Comment (CT)

<table>
<thead>
<tr>
<th>cid</th>
<th>cblog</th>
<th>cuser</th>
<th>msg</th>
<th>date</th>
</tr>
</thead>
<tbody>
<tr>
<td>t₁₁</td>
<td>c₀₁</td>
<td>b₀₁</td>
<td>u₀₁</td>
<td>Exactly what I was looking for!</td>
</tr>
</tbody>
</table>

Relational Schema Graph

- Construction of full schema paths

\[ sp_1 : FR.fuser \rightarrow US.uid \rightarrow US.uname. \]
\[ sp_2 : FR.fuser \rightarrow FR.fblog \rightarrow BG.bid \rightarrow BG.bname. \]
\[ sp_3 : FR.fuser \rightarrow FR.fblog \rightarrow BG.bid \rightarrow BG.admin \rightarrow US.uid \rightarrow US.uname. \]
\[ sp_4 : TG.tuser \rightarrow US.uid \rightarrow US.uname. \]
\[ sp_5 : TG.tuser \rightarrow TG.tcomment \rightarrow CT.cid \rightarrow CT.msg. \]
\[ sp_6 : TG.tuser \rightarrow TG.tcomment \rightarrow CT.cid \rightarrow CT.date. \]
\[ sp_7 : TG.tuser \rightarrow TG.tcomment \rightarrow CT.cid \rightarrow CT.cblog \rightarrow BG.bid \rightarrow BG.bname. \]
\[ sp_8 : TG.tuser \rightarrow TG.tcomment \rightarrow CT.cid \rightarrow CT.cuser \rightarrow US.uid \rightarrow US.uname. \]
\[ sp_9 : TG.tuser \rightarrow TG.tcomment \rightarrow CT.cid \rightarrow CT.cblog \rightarrow BG.bid \rightarrow BG.admin \rightarrow US.uid \rightarrow US.uname. \]
Property Graphs

- Walking the schema paths to create the property graph

\[ \begin{align*}
sp_1 : & \text{FR.fuser} \rightarrow \text{US.uid} \rightarrow \text{US.uname}. \\
sp_2 : & \text{FR.fuser} \rightarrow \text{FR.fblog} \rightarrow \text{BG.bid} \rightarrow \text{BG.bname}. \\
sp_3 : & \text{FR.fuser} \rightarrow \text{FR.fblog} \rightarrow \text{BG.bid} \rightarrow \text{BG.admin} \\
& \rightarrow \text{US.uid} \rightarrow \text{US.uname}. \\
sp_4 : & \text{TG.tuser} \rightarrow \text{US.uid} \rightarrow \text{US.uname}. \\
sp_5 : & \text{TG.tuser} \rightarrow \text{TG.tcomment} \rightarrow \text{CT.cid} \rightarrow \text{CT.msg}. \\
sp_6 : & \text{TG.tuser} \rightarrow \text{TG.tcomment} \rightarrow \text{CT.cid} \rightarrow \text{CT.date}. \\
sp_7 : & \text{TG.tuser} \rightarrow \text{TG.tcomment} \rightarrow \text{CT.cid} \rightarrow \text{CT.cblog} \\
& \rightarrow \text{BG.bid} \rightarrow \text{BG.bname}. \\
sp_8 : & \text{TG.tuser} \rightarrow \text{TG.tcomment} \rightarrow \text{CT.cid} \rightarrow \text{CT.cuser} \\
& \rightarrow \text{US.uid} \rightarrow \text{US.uname}. \\
sp_9 : & \text{TG.tuser} \rightarrow \text{TG.tcomment} \rightarrow \text{CT.cid} \rightarrow \text{CT.cblog} \\
& \rightarrow \text{BG.bid} \rightarrow \text{BG.admin} \rightarrow \text{US.uid} \rightarrow \text{US.uname}.
\end{align*} \]

- Read the paper
Query Transformation

• Find users who made comments on the ‘Information Systems’ blog
  • `select` US.uname
  • `from` User US, Tag TG, Blog BG, Comment CT
  • `where` (BG.bid = CT.cblog) and (CT.cid = TG.tcomment) and (TG.tuser = US.uid) and (BG.bname = 'Inf. Systems')