Lecture #7

Query Optimization
May 16th, 2002

Agenda/Administration

- Exam date set: June 10th, 6:30pm. Place TBA.
- Volunteers for presenting projects during last class.
- Project demos. Schedules coming soon.

Query Optimization

- Major issues:
  – Transformations (we saw a few, more coming)
    - Un-nesting of subqueries; magic-set transformations.
  – Join ordering
  – Maintaining statistics
  – General architectural issues to deal with large search space.

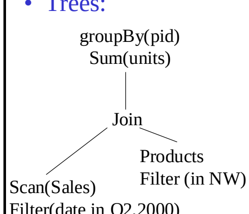
Schema for Some Examples

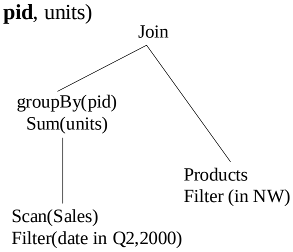
Sailors (<u>*sid*: integer</u>, *sname*: string, *rating*: integer, *age*: real)
Reserves (<u>*sid*: integer, *bid*: integer, *day*: dates</u>, *rname*: string)

- Reserves:
  – Each tuple is 40 bytes long, 100 tuples per page, 1000 pages (4000 tuples)
- Sailors:
  – Each tuple is 50 bytes long, 80 tuples per page, 500 pages (4000 tuples).

Rewrites: Group By and Join

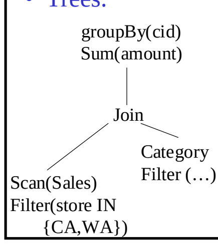
- Schema:
  – Product (**pid**, unitprice,…)
  – Sales(tid, date, store, **pid**, units)
- Trees:

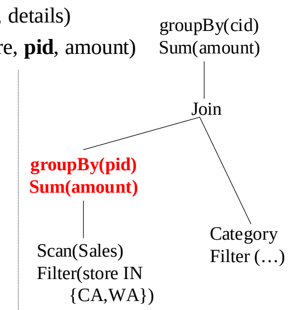




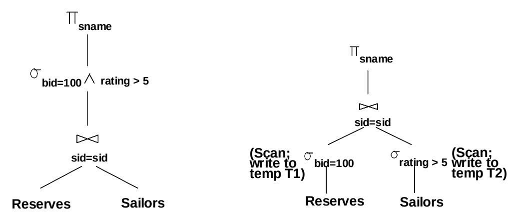
Rewrites:Operation Introduction

- Schema: (pid determines cid)
  – Category (**pid**, cid, details)
  – Sales(tid, date, store, **pid**, amount)
- Trees:





Query Rewriting: Predicate Pushdown



The earlier we process selections, less tuples we need to manipulate higher up in the tree.

Disadvantages?

Query Rewrites: Predicate Pushdown (through grouping)

```
Select   bid, Max(age)
From     Reserves R, Sailors S
Where    R.sid=S.sid
GroupBy  bid
Having   Max(age) > 40
```

```
Select   bid, Max(age)
From     Reserves R, Sailors S
Where    R.sid=S.sid  and
         S.age > 40
GroupBy  bid
```

- *For each boat, find the maximal age of sailors who've reserved it.*
- Advantage: the size of the join will be smaller.
- Requires transformation rules specific to the grouping/aggregation operators.
- **Will it work work if we replace Max by Min?**

Query Rewrite: Predicate Movearound

Sailing wiz dates: when did the youngest of each sailor level rent boats?

```
Select   sid, date
From     V1, V2
Where    V1.rating = V2.rating  and
         V1.age = V2.age
```

```
Create View V1 AS
Select   rating, Min(age)
From     Sailors S
Where    S.age < 20
Group By  rating
```

```
Create View V2 AS
Select   sid, rating, age, date
From     Sailors S, Reserves R
Where    R.sid=S.sid
```

Query Rewrite: Predicate Movearound

Sailing wiz dates: when did the youngest of each sailor level rent boats?

First, move predicates up the tree.

```
Select   sid, date
From     V1, V2
Where    V1.rating = V2.rating  and
         V1.age = V2.age, age < 20
```

```
Create View V1 AS
Select   rating, Min(age)
From     Sailors S
Where    S.age < 20
Group By  rating
```

```
Create View V2 AS
Select   sid, rating, age, date
From     Sailors S, Reserves R
Where    R.sid=S.sid
```

Query Rewrite: Predicate Movearound

Sailing wiz dates: when did the youngest of each sailor level rent boats?

First, move predicates up the tree.

Then, move them down.

```
Select   sid, date
From     V1, V2
Where    V1.rating = V2.rating  and
         V1.age = V2.age, and age < 20
```

```
Create View V1 AS
Select   rating, Min(age)
From     Sailors S
Where    S.age < 20
Group By  rating
```

```
Create View V2 AS
Select   sid, rating, age, date
From     Sailors S, Reserves R
Where    R.sid=S.sid, and
         S.age < 20.
```

Query Rewrite Summary

- The optimizer can use any *semantically correct* rule to transform one query to another.
- Rules try to:
  - move constraints between blocks (because each will be optimized separately)
  - Unnest blocks
- Especially important in decision support applications where queries are very complex.
- In a few minutes of thought, you'll come up with your own rewrite. Some query, somewhere, will benefit from it.
- Theorems?

Cost Estimation

- For each plan considered, must estimate cost:
  - Must estimate *cost* of each operation in plan tree.
    - Depends on input cardinalities.
  - Must estimate *size of result* for each operation in tree!
    - Use information about the input relations.
    - For selections and joins, assume independence of predicates.
- We'll discuss the System R cost estimation approach.
  - Very inexact, but works ok in practice.
  - More sophisticated techniques known now.

Cost Estimation

- What statistics should we save?

- How should we estimate the size of a query of the form?

```
SELECT attribute list
FROM relation list
WHERE term_1 AND ... AND term_k
```

Statistics and Catalogs

- Need information about the relations and indexes involved. ***Catalogs*** typically contain at least:
  - # tuples (NTuples) and # pages (NPages) for each relation.
  - # distinct key values (NKeys) and NPages for each index.
  - Index height, low/high key values (Low/High) for each tree index.
- Catalogs updated periodically.
  - Updating whenever data changes is too expensive; lots of approximation anyway, so slight inconsistency ok.
- More detailed information (e.g., histograms of the values in some field) are sometimes stored.

Cost Model for Our Analysis

- As a good approximation, we ignore CPU costs:
  - **B:** The number of data pages
  - **P:** Number of tuples per page
  - **D:** (Average) time to read or write disk page
  - Measuring number of page I/O's ignores gains of pre-fetching blocks of pages; thus, even I/O cost is only approximated.

Size Estimation and Reduction Factors

- Consider a query block:

```
SELECT attribute list
FROM relation list
WHERE term_1 AND ... AND term_k
```

- Maximum # tuples in result is the product of the cardinalities of relations in the FROM clause.
- *Reduction factor (RF)* associated with each *term* reflects the impact of the *term* in reducing result size. *Result cardinality* = Max # tuples * product of all RF's.
  - Implicit assumption that *terms* are independent!
  - Term *col=value* has RF *1/NKeys(I),* given index I on *col*
  - Term *col1=col2* has RF *1/MAX(NKeys(I1), NKeys(I2))*
  - Term *col>value* has RF *(High(I)-value)/(High(I)-Low(I))*

Histograms

- Key to obtaining good cost and size estimates.
- Come in several flavors:
  - Equi-depth
  - Equi-width
- Which is better?
- Compressed histograms: special treatment of frequent values.

Histograms

Employee(ssn, name, salary, phone)

- Maintain a histogram on salary:

Salary:	0..20k	20k..40k	40k..60k	60k..80k	80k..100k	> 100k
Tuples	200	800	5000	12000	6500	500

- T(Employee) = 25000, but now we know the distribution

Histograms

Ranks(rankName, salary)

- Estimate the size of Employee $\bowtie_{Salary}$ Ranks

Employee	0..20k	20k..40k	40k..60k	60k..80k	80k..100k	> 100k
	200	800	5000	12000	6500	500

Ranks	0..20k	20k..40k	40k..60k	60k..80k	80k..100k	> 100k
	8	20	40	80	100	2

Histograms

- Assume:
  - V(Employee, Salary) = 200
  - V(Ranks, Salary) = 250
- Then T(Employee $\bowtie_{Salary}$ Ranks) =

  $= \Sigma_{i=1,6} T_i T_i' / 250$

  $= (200\text{x}8 + 800\text{x}20 + 5000\text{x}40 + 12000\text{x}80 + 6500\text{x}100 + 500\text{x}2)/250$

  = ….

Plans for Single-Relation Queries (Prep for Join ordering)

- **Task:** create a query execution plan for a single Select-project-group-by block.
- **Key idea**: consider each possible ***access path*** to the relevant tuples of the relation. Choose the cheapest one.
- The different operations are essentially carried out together (e.g., if an index is used for a selection, projection is done for each retrieved tuple, and the resulting tuples are *pipelined* into the aggregate computation).

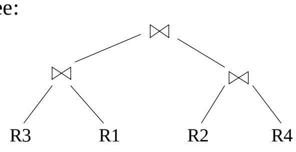
Example

```
SELECT S.sid
FROM Sailors S
WHERE S.rating=8
```

- If we have an Index on *rating*:
  - (1/NKeys(I)) * NTuples(R) = (1/10) * 40000 tuples retrieved.
  - Clustered index: (1/NKeys(I)) * (NPages(I)+NPages(R)) = (1/10) * (50+500) pages are retrieved (= 55).
  - Unclustered index: (1/NKeys(I)) * (NPages(I)+NTuples(R)) = (1/10) * (50+40000) pages are retrieved.
- If we have an index on *sid*:
  - Would have to retrieve all tuples/pages. With a clustered index, the cost is 50+500.
- Doing a file scan: we retrieve all file pages (500).

Determining Join Ordering

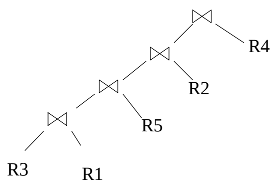
- R1 $\bowtie$ R2 $\bowtie$ …. $\bowtie$ Rn
- Join tree:



- A join tree represents a plan. An optimizer needs to inspect many (all ?) join trees

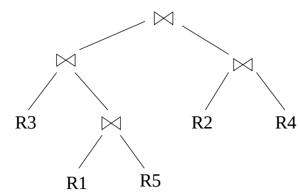
Types of Join Trees

- Left deep:



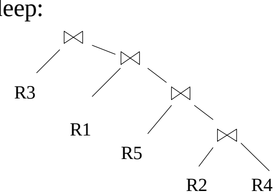
Types of Join Trees

- Bushy:



Types of Join Trees

- Right deep:



Problem

- Given: a query R1 $\bowtie$ R2 $\bowtie$ … $\bowtie$ Rn
- Assume we have a function cost() that gives us the cost of every join tree
- Find the best join tree for the query
- How?

Dynamic Programming

- Idea: for each subset of {R1, …, Rn}, compute the best plan for that subset
- In increasing order of set cardinality:
  - Step 1: for {R1}, {R2}, …, {Rn}
  - Step 2: for {R1,R2}, {R1,R3}, …, {Rn-1, Rn}
  - …
  - Step n: for {R1, …, Rn}
- A subset of {R1, …, Rn} is also called a *subquery*

Dynamic Programming

- For each subquery Q $\subseteq$ {R1, …, Rn} compute the following:
  - Size(Q)
  - A best plan for Q: Plan(Q)
  - The cost of that plan: Cost(Q)

Dynamic Programming

- **Step 1**: For each {Ri} do:
  - Size({Ri}) = B(Ri)
  - Plan({Ri}) = Ri
  - Cost({Ri}) = (cost of scanning Ri)

Dynamic Programming

- **Step i**: For each Q ⊆ {R1, …, Rn} of cardinality i do:
  - Compute Size(Q) (later…)
  - For every pair of subqueries Q’, Q’’ s.t. Q = Q’ U Q’’ compute cost(Plan(Q’) ⋈ Plan(Q’’))
  - Cost(Q) = the smallest such cost
  - Plan(Q) = the corresponding plan

Dynamic Programming

- Return Plan({R1, …, Rn})

Dynamic Programming

- Summary: computes optimal plans for subqueries:
  - Step 1: {R1}, {R2}, …, {Rn}
  - Step 2: {R1, R2}, {R1, R3}, …, {Rn-1, Rn}
  - …
  - Step n: {R1, …, Rn}
- We used naïve size/cost estimations
- In practice:
  - more realistic size/cost estimations (next)
  - heuristics for Reducing the Search Space
    - Restrict to left linear trees
    - Restrict to trees “without cartesian product”
  - need more than just one plan for each subquery:
    - “interesting orders”
- Why did it work?

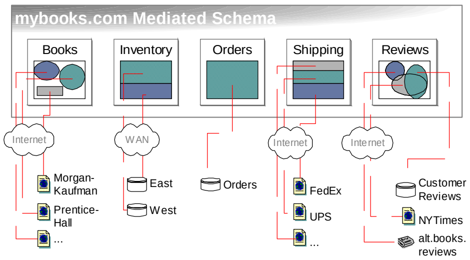
Query Optimization

- We’re done.
- Questions? Comments?

What is Data Integration

- Providing
  - Uniform (same query interface to all sources)
  - Access to (queries; eventually updates too)
  - Multiple (we want many, but 2 is hard too)
  - Autonomous (DBA doesn’t report to you)
  - Heterogeneous (data models are different)
  - Structured (or at least semi-structured)
  - Data Sources (not only databases).

The Problem: Data Integration

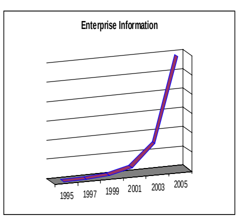


Uniform query capability across *autonomous*, *heterogeneous* data sources on LAN, WAN, or Internet

Motivation(s)

- Enterprise data integration; web-site construction.
- WWW:
  - Comparison shopping
  - Portals integrating data from multiple sources
  - B2B, electronic marketplaces
- Science and culture:
  - Medical genetics: integrating genomic data
  - Astrophysics: monitoring events in the sky.
  - Environment: Puget Sound Regional Synthesis Model
  - Culture: uniform access to all cultural databases produced by countries in Europe.

And if that wasn't enough…



Source: Gartner, 1999

- Explosion of intranet and extranet information
- 80% of corporate information is unmanaged
- By 2004 30X more enterprise data than 1999
- The average company:
  - maintains 49 distinct enterprise applications
  - spends 35% of total IT budget on integration-related efforts

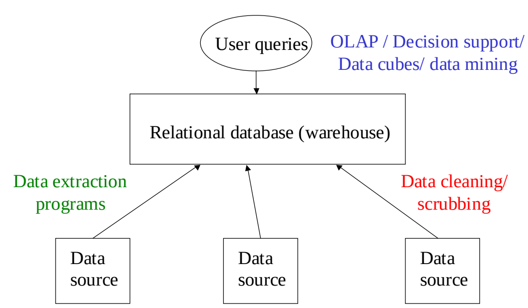
Discussion

- Why is it hard?
- How will we solve it?

Current Solutions

- Mostly ad-hoc programming: create a special solution for every case; pay consultants a lot of money.
- Data warehousing: load all the data periodically into a warehouse.
  - 6-18 months lead time
  - Separates *operational* DBMS from *decision support* DBMS. (not only a solution to data integration).
  - Performance is good; data may not be fresh.
  - Need to clean, scrub you data.

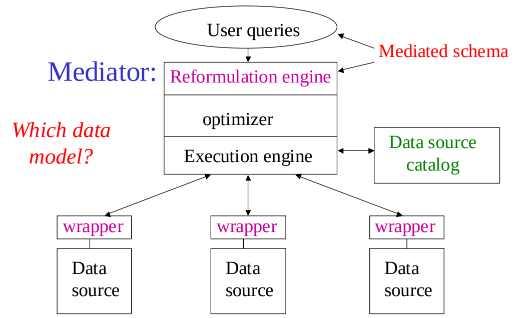
Data Warehouse Architecture



The Virtual Integration Architecture

- Leave the data in the sources.
- When a query comes in:
  - Determine the relevant sources to the query
  - Break down the query into sub-queries for the sources.
  - Get the answers from the sources, and combine them appropriately.
- Data is fresh.
- Challenge: performance.

Virtual Integration Architecture



Sources can be: relational, hierarchical (IMS), structure files, web sites.

Research Projects

- Garlic (IBM),
- Information Manifold (AT&T)
- Tsimmis, InfoMaster (Stanford)
- The Internet Softbot/Razor/Tukwila (UW)
- Hermes (Maryland)
- DISCO (INRIA, France)
- SIMS/Ariadne (USC/ISI)

Dimensions to Consider

- How many sources are we accessing?
- How autonomous are they?
- Meta-data about sources?
- Is the data structured?
- Queries or also updates?
- Requirements: accuracy, completeness, performance, handling inconsistencies.
- Closed world assumption vs. open world?

Outline

- Wrappers
- Semantic integration and source descriptions:
  - Modeling source completeness
  - Modeling source capabilities
- Query optimization
- Query execution (mostly Zack)

Wrapper Programs

- Task: to communicate with the data sources and do format translations.
- They are built w.r.t. a specific source.
- They can sit either at the source or at the mediator.
- Often hard to build (very little science).
- Can be “intelligent”: perform source-specific optimizations.

Example

Transform:

 Introduction to DB
<i> Phil Bernstein </i>
<i> Eric Newcomer </i>
Addison Wesley, 1999

into:

<book>
<title> Introduction to DB </title>
<author> Phil Bernstein </author>
<author> Eric Newcomer </author>
<publisher> Addison Wesley </publisher>
<year> 1999 </year>
</book>

Data Source Catalog

- Contains all meta-information about the sources:
  - Logical source contents (books, new cars).
  - Source capabilities (can answer SQL queries)
  - Source completeness (has *all* books).
  - Physical properties of source and network.
  - Statistics about the data (like in an RDBMS)
  - Source reliability
  - Mirror sources
  - Update frequency.

Content Descriptions

- User queries refer to the *mediated schema*.
- Data is stored in the sources in a *local schema*.
- Content descriptions provide the semantic mappings between the different schemas.
- Data integration system uses the descriptions to translate user queries into queries on the sources.

Desiderata from Source Descriptions

- Expressive power: distinguish between sources with closely related data. Hence, be able to prune access to irrelevant sources.
- Easy addition: make it easy to add new data sources.
- Reformulation: be able to reformulate a user query into a query on the sources efficiently and effectively.

Reformulation Problem

- Given:
  - A query Q posed over the mediated schema
  - Descriptions of the data sources
- Find:
  - A query Q' over the data source relations, such that:
    - Q' provides only *correct answers* to Q, and
    - Q' provides *all* possible answers from to Q given the sources.

Approaches to Specifying Source Descriptions

- Global-as-view: express the mediated schema relations as a set of views over the data source relations
- Local-as-view: express the source relations as views over the mediated schema.
- Can be combined with no additional cost.

Global-as-View

Mediated schema:
  Movie(title, dir, year, genre),
  Schedule(cinema, title, time).
Create View Movie AS
  select * from S1 [S1(title,dir,year,genre)]
  union
  select * from S2 [S2(title, dir,year,genre)]
  union [S3(title,dir), S4(title,year,genre)]
  select S3.title, S3.dir, S4.year, S4.genre
  from S3, S4
  where S3.title=S4.title

Global-as-View: Example 2

Mediated schema:
  Movie(title, dir, year, genre),
  Schedule(cinema, title, time).

Create View Movie AS [S1(title,dir,year)]
  select title, dir, year, NULL
  from S1
  union [S2(title, dir,genre)]
  select title, dir, NULL, genre
  from S2

Global-as-View: Example 3

Mediated schema:
  Movie(title, dir, year, genre),
  Schedule(cinema, title, time).
Source S4: S4(cinema, genre)
Create View Movie AS
  select NULL, NULL, NULL, genre
  from S4
 Create View Schedule AS
  select cinema, NULL, NULL
  from S4.

But what if we want to find which cinemas are playing comedies?

Global-as-View Summary

- Query reformulation boils down to view unfolding.
- Very easy conceptually.
- Can build hierarchies of mediated schemas.
- You sometimes loose information. Not always natural.
- Adding sources is hard. Need to consider all other sources that are available.

Local-as-View: example 1

Mediated schema:
  Movie(title, dir, year, genre),
  Schedule(cinema, title, time).
Create Source S1 AS
  select * from Movie
Create Source S3 AS [S3(title, dir)]
  select title, dir from Movie
Create Source S5 AS
  select title, dir, year
  from Movie
  where year > 1960 AND genre="Comedy"

Local-as-View: Example 2

Mediated schema:
  Movie(title, dir, year, genre),
  Schedule(cinema, title, time).
Source S4: S4(cinema, genre)
Create Source S4
  select cinema, genre
  from Movie m, Schedule s
  where m.title=s.title
.

Now if we want to find which cinemas are playing comedies, there is hope!

Local-as-View Summary

- Very flexible. You have the power of the entire query language to define the contents of the source.
- Hence, can easily distinguish between contents of closely related sources.
- Adding sources is easy: they're independent of each other.
- Query reformulation: *answering queries using views!*

The General Problem

- Given a set of views V1,…,Vn, and a query Q, can we answer Q using only the answers to V1,…,Vn?
- Many, many papers on this problem.
- The best performing algorithm: The MiniCon Algorithm, (Pottinger & Levy, 2000).
- Great survey on the topic: (Halevy, 2000).

Local Completeness Information

- If sources are incomplete, we need to look at each one of them.
- Often, sources are *locally complete*.
- Movie(title, director, year) complete for years after 1960, or for American directors.
- Question: given a set of local completeness statements, is a query Q' a complete answer to Q?

Example

- Movie(title, director, year) (complete after 1960).
- Show(title, theater, city, hour)
- Query: find movies (and directors) playing in Seattle:

  Select m.title, m.director

  From Movie m, Show s

  Where m.title=s.title AND city="Seattle"
- Complete or not?

Example #2

- Movie(title, director, year), Oscar(title, year)
- Query: find directors whose movies won Oscars after 1965:

  select m.director

  from Movie m, Oscar o

  where m.title=o.title AND m.year=o.year AND o.year > 1965.
- Complete or not?

Query Optimization

- Very related to query reformulation!
- Goal of the optimizer: find a physical plan with minimal cost.
- Key components in optimization:
  - Search space of plans
  - Search strategy
  - Cost model

Optimization in Distributed DBMS

- A distributed database (2-minute tutorial):
  - Data is distributed over multiple nodes, but is uniform.
  - Query execution can be distributed to sites.
  - Communication costs are significant.
- Consequences for optimization:
  - Optimizer needs to decide locality
  - Need to exploit independent parallelism.
  - Need operators that reduce communication costs (semi-joins).

DDBMS vs. Data Integration

- In a DDBMS, data is distributed over a set of *uniform* sites with *precise* rules.
- In a data integration context:
  - Data sources may provide only limited access patterns to the data.
  - Data sources may have additional query capabilities.
  - Cost of answering queries at sources unknown.
  - Statistics about data unknown.
  - Transfer rates unpredictable.

Modeling Source Capabilities

- Negative capabilities:
  - A web site may require certain inputs (in an HTML form).
  - Need to consider only valid query execution plans.
- Positive capabilities:
  - A source may be an ODBC compliant system.
  - Need to decide placement of operations according to capabilities.
- Problem: *how to describe and exploit source capabilities.*

Example #1: Access Patterns

Mediated schema relation: Cites(paper1, paper2)

```
Create Source S1 as
  select *
  from Cites
  given paper1
Create Source S2 as
   select paper1
   from Cites
```

Query: select paper1 from Cites where paper2="Hal00"

Example #1: Continued

```
Create Source S1 as
  select *
  from Cites
  given paper1
Create Source S2 as
   select paper1
   from Cites
Select p1
From S1, S2
Where S2.paper1=S1.paper1 AND S1.paper2="Hal00"
```

Example #2: Access Patterns

```
Create Source S1 as
  select *
  from Cites
  given paper1
Create Source S2 as
   select paperID
   from UW-Papers
Create Source S3 as
    select paperID
    from  AwardPapers
    given paperID
```

Query: select * from AwardPapers

Example #2: Solutions

- Can't go directly to S3 because it requires a binding.
- Can go to S1, get UW papers, and check if they're in S3.
- Can go to S1, get UW papers, feed them into S2, and feed the results into S3.
- Can go to S1, feed results into S2, feed results into S2 again, and then feed results into S3.
- Strictly speaking, we can't a priori decide when to stop.
- Need *recursive query processing*.

Handling Positive Capabilities

- Characterizing positive capabilities:
  - Schema independent (e.g., can always perform joins, selections).
  - Schema dependent: can join R and S, but not T.
  - Given a query, tells you whether it can be handled.
- Key issue: how do you search for plans?
- Garlic approach (IBM): Given a query, STAR rules determine which subqueries are executable by the sources. Then proceed bottom-up as in System-R.

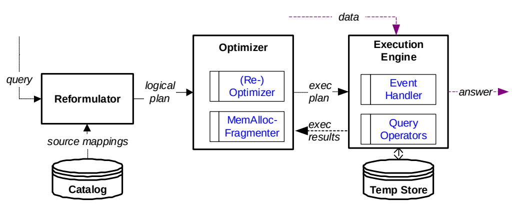
Matching Object Across Sources

- How do I know that A. Halevy in source 1 is the same as Alon Halevy in source 2?
- If there are uniform keys across sources, no problem.
- If not:
  - Domain specific solutions (e.g., maybe look at the address, ssn).
  - Use Information retrieval techniques (Cohen, 98). Judge similarity as you would between documents.
  - Use concordance tables. These are time-consuming to build, but you can then sell them for lots of money.

Optimization and Execution

- Problem:
  - Few and unreliable statistics about the data.
  - Unexpected (possibly bursty) network transfer rates.
  - Generally, unpredictable environment.
- General solution: (research area)
  - Adaptive query processing.
  - Interleave optimization and execution. As you get to know more about your data, you can improve your plan.

Tukwila Data Integration System



Novel components:
- Event handler
- Optimization-execution loop