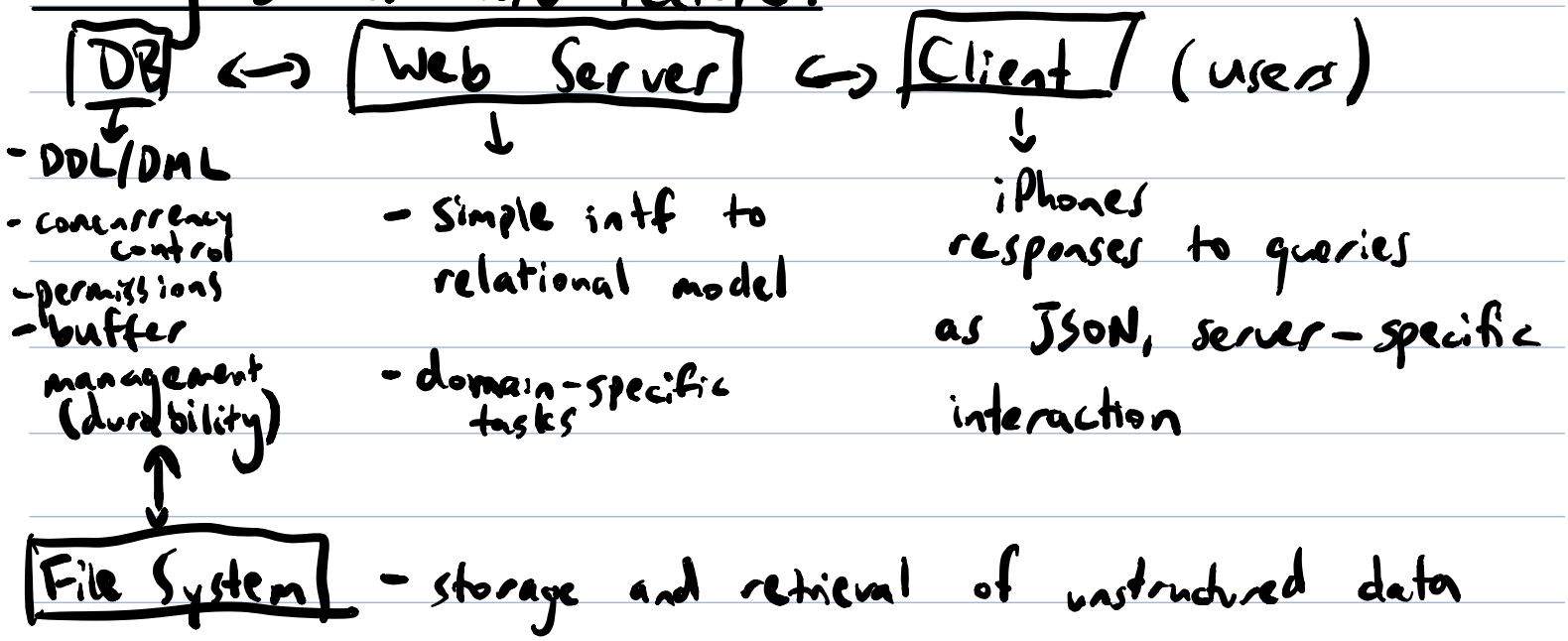
DB Tuning and Physical Design: Basics of Query Execution

We assume our tool uses a client-server architecture. Historically, db server code was implemented over a file system. In this course, assume a built-in ability to work with a 2-column table containing filenames and corresponding value of the file (sequence of bytes) allowing random access.

Usually 3-tier architecture:



Query ① Optimization/compilation → Output: Query plan ② Execution → Output: results (tables)

Straight to ② for query evaluation.

Goal: develop a simple relational calculator that answers range-restricted queries (bottom-up)

Considerations:
- physical storage (physical design)
- Compute answers to complex queries
- manage intermediate results

how to execute queries!

1. parsing, type checking, etc
2. SQL translated to relational algebra
3. optimization
   - ⇒ generates an effective query plan
   - ⇒ uses statistics collected about the stored data
4. plan execution
   - ⇒ access methods to access stored relations
   - ⇒ physical relational operators to combine relations

Need:
- query
- Conceptual info (tables / schema)
- physical design (which data structs?)
- characteristics of result data (how big?)
  - ⇒ statistically

Rel^a algebra: $(\cup, R_0, \ldots, R_k, \times, \sigma_{p_i}, \pi_{v_i}, v_i, -)$ [select] [project]

each R_i needs to be a name of a materialized view.

Projection

Definition

$$\pi_V(R) = \{(x_{i_1}, \ldots, x_{i_m}) \mid (x_1, \ldots, x_n) \in R \wedge i_j \in V\}$$

where V is an ordered list of column numbers.

Selection

Definition

$$\sigma_\varphi(R) = \{(x_1, \ldots, x_n) \mid (x_1, \ldots, x_n) \in R \wedge \varphi(x_1, \ldots, x_n)\}$$

Product

e.g. Account × Bank
(order matters!)

Can consider a simple VM implementation of the rel^ algebra (nested loop).

Union: both sides need to have the same # of columns

$\pi_{1,2}(\sigma_{2='CHK'}(Account)) \cup \pi_{1,2}(\sigma_{2='SAV'}(Account))$

Difference

e.g. $\pi_{1,4}(Account) - \pi_{1,4}(\sigma_{4,6}(Account \times Bank))$

accounts without banks?

If we had bag semantics, we could add a unary operator for unique results only (duplicate elimination)

Theorem (Codd)

For every D.I. R.C. query there is an equivalent relational algebra expression.

$RCtoRA(R_i(x_1, \ldots, x_k)) = R_i$

$RCtoRA(Q \wedge x_i = x_j) = \sigma_{i=j}(RCtoRA(Q))$

$RCtoRA(\exists x_i \cdot Q) = \pi_{FV(Q)-\{x_i\}}(RCtoRA(Q))$

...

How do we (mostly) avoid storing intermediate results?

⇒ Cursor open/fetch/close interface

- every implementation of an R.A. operator

1) produces answers this way

2) gets answers from children this way

We make at least one physical implementation per operator.

This DB implementation will be very slow!

1) use disk-based data structs for efficient searching (indexing)
2) use better algos to implement operators (SORTING, HASHING)
3) rewrite expression to equivalent, more efficient one

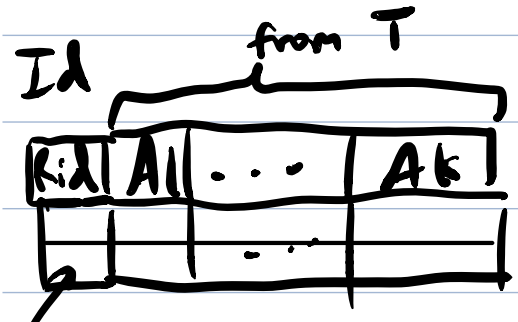
Atomic Relations and Indexing

- when an index $R_{index}(X)$ (where X is the search attribute) is available, we replace $\sigma_{X=c}(R)$ with accessing $R_{index}(X)$ directly
- otherwise, check all file blocks holding tuples for R
- even if idX available, searching entire tables may be faster if:
  - relation is small, or
  - rel^ is large, but we expect most of the tuples to satisfy the condition

Standard physical design

create table T ...

- instantly creates materialized view with an internally generated identifier
- create view Id as (select * from T) materialized



address of record encoding that tuple in the file

- this is automatically done
- vendor may provide commands to manipulate this physical design
  (create index ...)

create index I $(A_{i_1}, \ldots, A_{i_2})$ on T [clustered]

Clustering: tuples with similar value of attribute are stored together in the same block

e.g. create index Id() on T clustered
create index I1(A_3, A_4) on T

non-clustering (secondary): creates materialized view with columns (A_3, A_4, rid) where rid is a foreign key to the clustered index