

## 5. Physical Properties

- Why Properties
- Distributed Queries
- Ordering
- Grouping
- DAGs

# Why Properties

- query optimizer chooses the equivalent plan cheapest
- join ordering: the cheapest plan with the same set of relations
- but: plans might produce the same result but behave differently
- for example sort-merge vs. hash join
- hash join could be cheaper, but sort-merge still pay of later
- not directly comparable

## Why Properties (2)

How to handle logical equivalent but un-comparable plans?

- one alternative: encode differences into search space
- for example, different plans for sorting vs. hashing
- but: search space explodes
- some aspects like "sorting" consist of many alternatives
- further: if sorting is cheaper than hashing, we usually prefer sorting
- direct encoding into search space too wasteful
- use (physical) properties instead

## Using Properties

A physical property  $P$  defines a partial relation  $\leq_P$  with the following characteristics among plans:

If two plans  $p_1$  and  $p_2$  are logically equivalent,

- $p_1 \leq_P p_2$  if  $p_2$  dominates  $p_1$  concerning  $P$
- $p_1 =_P p_2$  is  $p_1$  and  $p_2$  are comparable concerning  $P$   
( $p_1 \leq_P p_2 \wedge p_2 \leq_P p_1$ )

A plan can only be pruned if it is dominated or comparable

## Using Properties (2)

With properties, the query optimizer does not maintain a single solution but a set of solutions for each subproblem:

```
storeSolution( $S, p$ )  
   $P = dpTable[S]$   
   $P' = \emptyset$   
  for  $\forall p' \in P$  {  
    if  $p \leq p' \wedge C(p) \geq C(p')$   
      return  
    if  $\neg(p' \leq p \wedge C(p') \geq C(p))$   
       $P' = P' \cup \{p'\}$   
  }  
   $dpTable[S] = P' \cup \{p\}$ 
```

## Using Properties (3)

- algorithm too simple
- properties can be *enforced*
- Enforcers make plans comparable
- allows for more pruning
- will see examples for this
- combination of multiple properties needs some care

# Distributed Queries

- distributed query processing keeps track of the *site*
- intermediate results can be computed at different sites
- a physical property is therefore the site of the intermediate result
- very simple property, site is either the same or different
- more plans comparable with enforcers

## Distributed Queries - Comparing Plans

Two plans are comparable, if they produce their result on the same site or the difference is larger than the shipment costs:

```
prune( $p_1, p_2$ )  
  if  $p_1.site = p_2.site$   
    return  $(C(p_1) \leq C(P_2)) ? p_1 : p_2$   
  if  $C(p_1) + C(\text{transfer } p_1) \leq C(P_2)$   
    return  $p_1$   
  if  $C(p_2) + C(\text{transfer } p_2) \leq C(P_1)$   
    return  $p_2$   
  return  $\{p_1, p_2\}$ 
```

## Distributed Queries - Effect on Search organization

- previous slide described how to compare plans, but not how to generate them
- plans must be generated for desired sites
- one possibility: generate plans for all sites
- can be quite wasteful
- alternative: generate plans (for sites) on demand
- difficult to do bottom-up
- usual technique: determine relevant sites beforehand and generate plans for them
- this sites would be called *interesting*

# Ordering

- physical tuple order is the classical physical property
- equivalent plans produce the same tuples, but (potentially) in different order
- tuple ordering is very important for many operators
- sort-merge, group by etc.
- explicit order by
- access optimization

## Ordering (2)

An ordering  $O$  is a list of attributes  $(A_1, \dots, A_n)$

A tuple stream satisfied an ordering  $O$ , if the tuples are sorted according to  $A_1$  and for each  $1 < i \leq n$  the tuples are sorted on  $A_i$  for identical values of  $A_1, \dots, A_{i-1}$ .

# Interesting Orderings

- optimizer uses existing orderings, or creates new ones (enforcers)
- set of potential orderings very large
- too many orderings increase the search space
- concentrate on relevant orderings: *interesting orderings*

ordering is interesting, if

- requested by the user
- physically available
- useful for a planed operator

## Interesting Orderings (2)

- ordering is characterized by a list of attributes
- if a tuple stream is ordered on  $a_1, \dots, a_n, a_{n+1}$ , it is also ordered on  $a_1, \dots, a_n$
- orderings are affected by operators, in particular they can grow
- therefore, each prefix of an interesting ordering is also interesting
- (somewhat implementation dependent)
- non-interesting orderings are "forgotten" by the optimizer to reduce the search space

# Physical vs. Logical Ordering

- the *physical* ordering is the actual order of tuples on disk/in a tuple stream
- the *logical* ordering is the ordering satisfied by the tuples
- the query optimizer can usually only reason about the logical ordering
- a tuple stream may satisfy multiple logical orderings
- the logical ordering can change, although the physical ordering did not!

# Functional Dependencies

Logical Ordering is affected by functional dependencies:

- induces by operators
- $\sigma_{a=\cos(b)} \Rightarrow \{b \rightarrow a\}$
- $\sigma_{a=b} \Rightarrow \{a \rightarrow b, b \rightarrow a\}$  (even stronger)
- $\sigma_{a=10} \Rightarrow \{\emptyset \rightarrow a\}$
- complex operators can induce multiple FDs
- FDs allow for deriving new logical orderings

## Example

```
select a,b,c
from   s a,
      (select b:b,c:count(*),d:max(d)
       from tablefunc(a) group by b)
order  by a,b,c
```

Interesting ordering:  $(a)$ ,  $(b)$ ,  $(a, b)$  and  $(a, b, c)$

Interesting groupings:  $\{b\}$

Functional dependencies:  $b \rightarrow c$ ,  $b \rightarrow d$

- Note: for  $\{b\}$  grouping is sufficient (next section)

# Materializing Orderings

- the query optimizer might just maintain a set of all orderings satisfied by a plan
- but FDs increase the set
- $\text{sort}(a) \rightarrow \text{select}(a = b)$
- is compatible with  $(a), (a, b), (b), (b, a)$
- set can grow exponentially
- maintaining set of orderings not feasible

# Reducing Orderings

Simmen et al. [9] proposed the following scheme:

- remember the base ordering
- remember all functional dependencies
- whenever testing for an ordering, reduce by base ordering and functional dependency
- apply prefix test after this

## Reducing Orderings - Example

Ordering  $(b, d, e)$ , test for  $(a, b, c, e)$ , FDs  $\{a \rightarrow c, \emptyset \rightarrow a, b \rightarrow d\}$

1. reduce ordering to  $(b, e)$
2. reduce test to  $(a, b, e)$
3. reduce test to  $(b, e)$
4. test for prefix

but:

- what would happen if we applied  $\emptyset \rightarrow a$  first?
- reductions must be applied back to front

## Reducing Orderings - Discussion

- back-to-front rule is not enough  $((a), (a, b, c), \{a \rightarrow b, a, b \rightarrow c\})$
- avoiding this requires normalizing the FDs, which is very expensive
- reduction has to be done for each test
- tests happen very frequently (nearly each operator tests)
- memory management is a problem
- better than materializing orderings, but not optimal

# Required Interface for Orderings

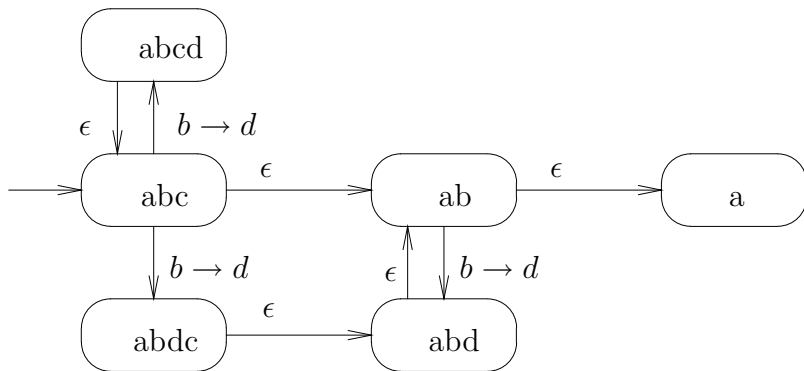
Query optimizer just requires few operations:

- initialization
- test for an ordering
- apply function dependency

Concrete ordering not required

# Encoding Orderings as FSMs

Use an FSM (ordering  $(a, b, c)$ , FD  $\{b \rightarrow d\}$ )

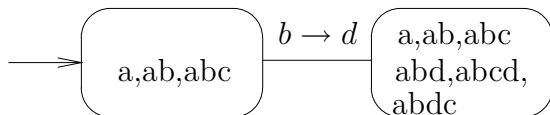


## Encoding Orderings as FSMs (2)

- FSM described physical orderings
- pretends that FD changes physical ordering
- might be non-deterministic
- has to become deterministic
- conversion in DFSM (via  $NFA \rightarrow DFA$ )

## Encoding Orderings as FSMs (3)

DFSM



- node contains all possible physical orderings  $\implies$  logical orderings
- operating on the DFSM is very efficient
- only problem: how to construct it (efficiently)

# Ordering FSM Construction - Overview

1. Determine the input
  - 1.1 Determine interesting orders
  - 1.2 Determine sets of functional dependencies
2. Construct the NFSM
  - 2.1 Construct nodes of the NFSM
  - 2.2 Filter functional dependencies
  - 2.3 Add edges to the NFSM
  - 2.4 Prune the NFSM
  - 2.5 Add artificial start node and edges
3. Construct the DFSM - convert the NFSM into a DFSM
4. Precompute values
  - 4.1 Precompute the compatibility matrix
  - 4.2 Precompute the transition table

# Ordering FSM Construction - Determining the Input

- interesting orders (requested, required, index)
- $O_I = O_P \cup O_T$  (produced vs. tested, allows pruning)
- functional dependencies (operators, keys)
- handles for  $O(1)$  comparisons

E.g.

$$\mathcal{F} = \{\{b \rightarrow c\}, \{b \rightarrow d\}\}$$

$$O_I = \{(b), (a, b)\} \cup \{(a, b, c)\}$$

# Ordering FSM Construction - Constructing the NFSM

Initial nodes for  $O_I$

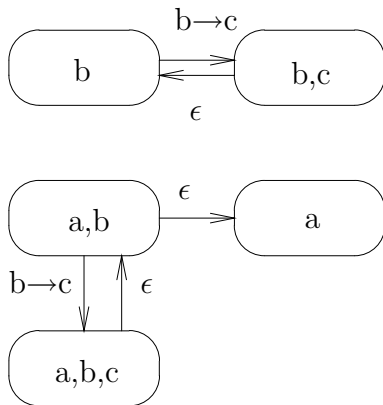
b

a,b

a,b,c

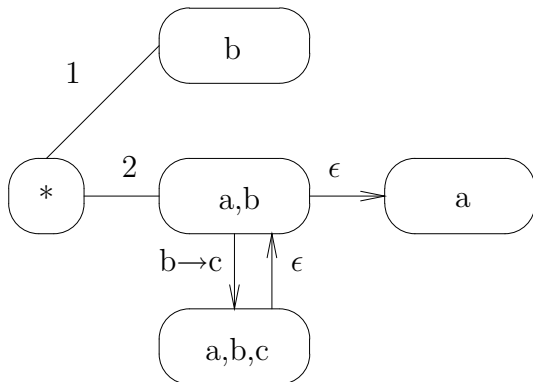
# Ordering FSM Construction - Constructing the NFSM (2)

Edges for  $F$ . Creates artificial node (can be pruned)



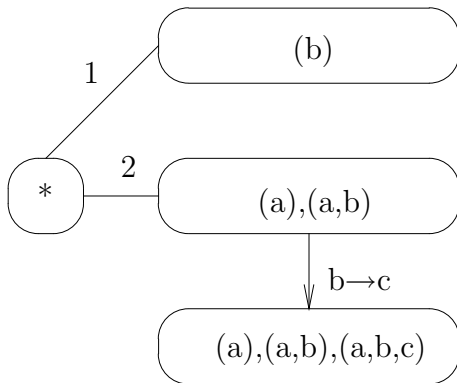
## Ordering FSM Construction - Constructing the NFSM (3)

Edges for initialization.  $(b, c)$  was pruned.



# Ordering FSM Construction - Constructing the DFSM

Standard conversion algorithm



- tests for  $O_T$  are precomputed (materialized)

# Pruning Techniques

- reducing the NFSM reduces conversion time
- reducing the DFSM reduces search space
- FDs can be removed if no interesting orderings reachable
- artificial nodes can be merged if they behave identically
- artificial nodes can be removed if they only have  $\epsilon$  edges

Note: search space reduction is a major benefit!

# Discussion

- orderings essential for query optimizations
- but orderings increase the search space
- management involved
- FSM representation needs  $O(1)$  time and space during optimization
- queried very often, but also very fast
- help reduce the search space