Big DataBase

wu

February 19, 2025

Contents

1 Query Optimization **2**
- 1.1 Introduction . . . 2
- 1.2 Query Optimization . . . 4
  - 1.2.1 Algebra Revisited . . . 4
  - 1.2.2 Canonical Query Translation . . . 6
  - 1.2.3 Logical Query Optimization . . . 7
  - 1.2.4 Physical Query Optimization . . . 7
- 1.3 Join Ordering . . . 8
  - 1.3.1 Basics . . . 8
  - 1.3.2 Search Space . . . 11
  - 1.3.3 Greedy Heuristics . . . 13
  - 1.3.4 IKKBZ . . . 13
  - 1.3.5 The Maximum-Value-Precedence Algorithm . . . 19
  - 1.3.6 Dynamic Programming . . . 20
  - 1.3.7 Simplifying the Query Graph . . . 20
  - 1.3.8 Adaptive Optimization . . . 20
  - 1.3.9 Generating Permutations . . . 20
  - 1.3.10 Transformative Approaches . . . 20
  - 1.3.11 Randomized Approaches . . . 20
  - 1.3.12 Metaheuristics . . . 20
  - 1.3.13 Iterative Dynamic Programming . . . 20
  - 1.3.14 Order Preserving Joins . . . 20
  - 1.3.15 Complexity of Join Processing . . . 20
- 1.4 Accessing the Data . . . 20
- 1.5 Physical Properties . . . 20
- 1.6 Query Rewriting . . . 20
- 1.7 Self Tuning . . . 20

2 Transaction System **20**
2.1 Computational Models 20
2.1.1 Page Model 20
2.1.2 Object Model 20
2.2 Notions of Correctness for the Page Model 21
2.2.1 Canonical Synchronization Problems 21
2.2.2 Syntax of Histories and Schedules 22
2.2.3 Herbrand Semantics of Schedules 23
2.2.4 Final-State Serializability 24
2.2.5 View Serializability 26
2.2.6 Conflict Serializability 27
2.2.7 Commit Serializability 29
2.2.8 An Alternative Criterion: Interleaving Specifications 29
2.3 Concurrency Control Algorithms 29
2.3.1 General Scheduler Design 29
2.3.2 Locking Schedulers 30
2.3.3 Non-Locking Schedulers 34
2.3.4 Hybrid Protocols 34
2.4 Multiversion Concurrency Control 34
2.4.1 Multiversion Schedules 34
2.4.2 Multiversion Serializability 35
2.4.3 Limiting the Number of Versions 35
2.4.4 Multiversion Concurrency Control Protocols 35

3 OLAP **35**
3.1 Columar store 35

1 Query Optimization

1.1 Introduction

Compile time system:

1. parsing: parsing, AST production
2. semantic analysis: schema lookup, variable binding, type inference
3. normalization, factorization, constant folding
4. rewrite 1: view resolution, unnesting, deriving predicates
5. plan generation: constructing the execution plan

6. rewrite 2: refining the plan, pushing group
7. code generation: producing the imperative plan

Different optimization goals:

- minimize response time
- minimize resource consumption
- minimize time to first tuple
- maximize throughput

Notation:

- $\mathcal{A}(e)$: attributes of the tuples produces by e
- $\mathcal{F}(e)$ free variable of the expression e
- binary operators $e_1 \theta e_2$ usually require $\mathcal{A}(e_1) = \mathcal{A}(e_2)$
- $\rho_{a \to b(e)}$, rename
- $\Pi_A(e)$, projection
- $\sigma_p(e)$, selection, $\{x \mid x \in e \wedge p(x)\}$
- $e_1 \bowtie_p e_2$, join, $\{x \circ y \mid x \in e_1 \wedge y \in e_2 \wedge p(x \circ y)\}$

Different join implementations have different characteristics:

- $e_1 \bowtie^{NL} e_2$ Nested Loop Join:
- $e_1 \bowtie^{BNL} e_2$ Blockwise Nested Loop Join: Read chunks of e_1 into memory and read e_2 once for each chunk. Further improvement: Use hashing for equi-joins
- $e_1 \bowtie^{SM} e_2$ Sort Merge Join: Equi-joins only
- $e_1 \bowtie^{HH} e_2$ Hybrid-Hash Join: Partitions e_1 and e_2 into partitions that can be joined in memory. Equi-joins only

1.2 Query Optimization

steps

1. translate the query into its canonical algebraic expression
2. logical query optimization
3. physical query optimization

1.2.1 Algebra Revisited

Tuple is a (unordered) mapping from attribute names to values of a domain
Schema is a set of attributes with domain, written $\mathcal{A}(t)$
concatenation of tuple:

- $t_1 \circ t_2$, note $t_1 \circ t_2 = t_2 \circ t_1$
- $\mathcal{A}(t_1) \cap \mathcal{A}(t_2) = \emptyset$
- $\mathcal{A}(t_1 \circ t_2) = \mathcal{A}(t_1) \cup \mathcal{A}(t_2)$

tuple projection:

- $t.a$, $t|_A$
- $a \in \mathcal{A}(t), A \subseteq \mathcal{A}(t)$
- $\mathcal{A}(t|_A) = A$
- $t.a$ produces a value, $t|_A$ produces a tuple

Relation is a set of tuples with the same schema. Schema of the contained tuples, written $\mathcal{A}(R)$

Real data is usually a multi set (bag). The optimizar must consider three different semantics:

- logical algebra operates on bags
- physical algebra operates on streams
- explicit duplicate elimination $\Rightarrow$ sets

Set operations are part of the algebra:

- union, intersection, difference

- but have schema constraints
- $\mathcal{A}(L) = \mathcal{A}(R)$
- $\mathcal{A}(L \cup R) = \mathcal{A}(L) = \mathcal{A}(R)$, $\mathcal{A}(L \cup R) = \mathcal{A}(L) = \mathcal{A}(R)$, $\mathcal{A}(L \setminus R) = \mathcal{A}(L) = \mathcal{A}(R)$

$\mathcal{F}(e)$ are the free variables of e
Selection:

- $\sigma_p(R)$
- $\mathcal{F}(p) \subseteq \mathcal{A}(R)$
- $\mathcal{A}(\sigma_p(R)) = \mathcal{A}(R)$

Projection:

- $\Pi_A(R)$
- eliminates duplicates for set semantic, keeps them for bag semantic
- $A \subseteq \mathcal{A}(R)$
- $\mathcal{A}(\Pi_A(R)) = A$

Rename:

- $\rho_{a \to b}(R)$
- $a \in \mathcal{A}(R), b \notin \mathcal{A}(R)$
- $\mathcal{A}(\rho_{a \to b}(R)) = \mathcal{A}(R) \setminus \{a\} \cup \{b\}$

$$
\begin{aligned}
\sigma_{p_1 \wedge p_2} &\equiv \sigma_{p_1}(\sigma_{p_2}(e)) && (1)\\
\sigma_{p_1}(\sigma_{p_2}(e)) &\equiv \sigma_{p_2}(\sigma_{p_1}(e)) && (2)\\
\Pi_{A_1}(\Pi_{A_2}(e)) &\equiv \Pi_{A_1}(e) && (3)\\
&\equiv \text{if } A_1 \subseteq A_2 &&\\
\sigma_p(\Pi_A(e)) &\equiv \Pi_A(\sigma_p(e)) && (4)\\
&\equiv \text{if } \mathcal{F}(p) \subseteq A &&\\
\sigma_p(e_1 \cup e_2) &\equiv \sigma_p(e_1) \cup \sigma_p(e_2) && (5)\\
\sigma_p(e_1 \cap e_2) &\equiv \sigma_p(e_1) \cap \sigma_p(e_2) && (6)\\
\sigma_p(e_1 \setminus e_2) &\equiv \sigma_p(e_1) \setminus \sigma_p(e_2) && (7)\\
\Pi_A(e_1 \cup e_2) &\equiv \Pi_A(e_1) \cup \Pi_A(e_2) && (8)\\
e_1 \times e_2 &\equiv e_2 \times e_1 && (9)\\
e_1 \bowtie_p e_2 &\equiv e_2 \bowtie_p e_1 && (10)\\
(e_1 \times e_2) \times e_3 &\equiv e_1 \times (e_2 \times e_3) && (11)\\
(e_1 \bowtie_{p_1} e_2) \bowtie_{p_2} e_3 &\equiv e_1 \bowtie_{p_1} (e_2 \bowtie_{p_2} e_3) && (12)\\
\sigma_p(e_1 \times e_2) &\equiv e_1 \bowtie_p e_2 && (13)\\
\sigma_p(e_1 \times e_2) &\equiv \sigma_p(e_1) \times e_2 && (14)\\
&\equiv \text{if } \mathcal{F}(e) \subseteq \mathcal{A}(e_1) &&\\
\sigma_{p_1}(e_1 \bowtie_{p_2} e_2) &\equiv \sigma_{p_1}(e_1) \bowtie_{p_2} e_2 && (15)\\
&\equiv \text{if } \mathcal{F}(p_1) \subseteq \mathcal{A}(e_1) &&\\
\Pi_A(e_1 \times e_2) &\equiv \Pi_{A_1}(e_1) \times \Pi_{A_2}(e_2) && (16)\\
&\equiv \text{if } A = A_1 \cup A_2, A_1 \subseteq \mathcal{A}(e_1), A_2 \subseteq \mathcal{A}(e_2) &&
\end{aligned}
$$

1.2.2 Canonical Query Translation

Restrictions:

- only **select distinct**
- no **group by, order by, union, intersect, except**
- only attributes in **select** clause
- no nested queries
- not discussed here: `NULL` values

1.2.3 Logical Query Optimization

- foundation: algebraic equivalence

Which plans are better?

- plans can only be compared if there is a cost function
- cost functions need details that are not available when only considering logical algebra
- consequence: logical query optimization remains a heuristic

Phases

1. break up conjunctive selection predicates, $(1) \rightarrow$
2. push selections down, $(2) \rightarrow, (14) \rightarrow$
3. introduce joins, $(13) \rightarrow$
4. determine join order $(9), (10), (11), (12)$
5. introduce and push down projections $(3) \leftarrow, (4) \leftarrow, (16) \rightarrow$
    - eliminate redundant attributes

This kind of phases has limitation: different join order would allow further push down. The phases are interdependent

1.2.4 Physical Query Optimization

- add more execution information to the plan
- allow for cost calculations
- select index structures/access paths
    - scan+selection could be done by an index lookup
    - multiple indices to choose from
    - table scan might be the best, even if an index is available
    - depends on selectivity, rule of thumb: 10%
    - detailed statistics and costs required
    - related problem: materialized view

- even more complex, as more than one operator could be substitued

- choose operator implementations
  - replace a logical operator (e.g. $\bowtie$) with a physical one (e.g. $\bowtie^{HH}$)
  - semantic restrictions: e.g., most join operators require equi-conditions
  - $\bowtie^{BNL}$ is better than $\bowtie^{NL}$
  - $\bowtie^{SM}$ and $\bowtie^{HH}$ are usually better than both
  - $\bowtie^{HH}$ is often the best if not reusing sorts
  - decision must be cost-based
  - even $\bowtie^{NL}$ can be optimal
  - not only joins, has to be done for all operators
- add property enforcer
  - certain physical operators need certain properties
  - example: sort for $\bowtie^{SM}$
  - example: in a distributed database, operators need the data locally to operate
  - many operator requirements can be modeled as properties
- choose when to materialize
  - temp operator stores input on disk
  - essential for multiple consumers (factorization, DAGs)
  - also relevant for $\bowtie^{NL}$

1.3 Join Ordering

1.3.1 Basics

Concentrate on join ordering, that is:

- conjunctive queries
- simple predicates
- predicates have the form $a_1 = a_2$ where a_1 is an attribute and a_2 is either an attribute or a constant

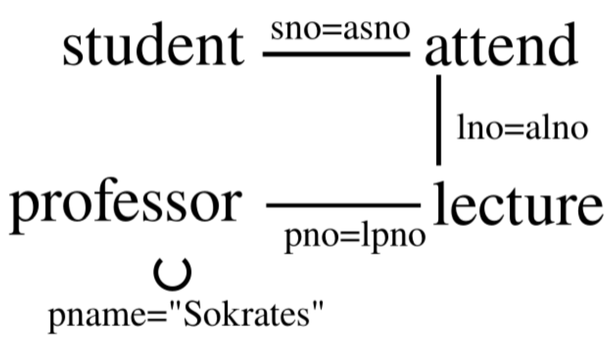
- even ignore constants in some algorithms

We join relations $R_1, \dots, R_n$ where R_i can be

- a base relation
- a base relation including selections
- a more complex building block or access path

Queries of this type can be characterized by their query graph:

- the query graph is an undirected graph with $R_1, \dots, R_n$ as nodes
- a predicate of the form $a_1 = a_2$ where $a_1 \in R_i$ and $a_2 \in R_j$ forms an edge between R_i and R_j labeled with the predicate
- a predicate of the form $a_1 = a_2$ where $a_1 \in R_i$ and a_2 is a constant forms a self-edge on R_i labeled with the predicate



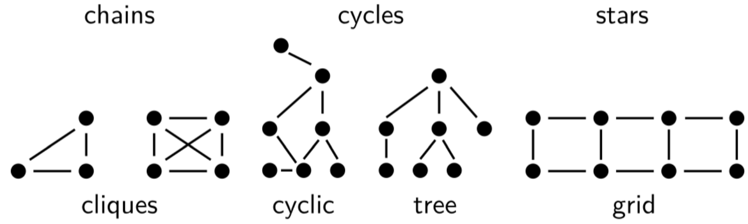


Figure 1: Shapes of Query Graphs

A join tree is a binary tree with

- join operators as inner nodes
- relations as leaf nodes

Commonly used classes of join trees:

- left-deep tree
- right-deep tree
- zigzag tree: at least one input of every join is a relation R
- bushy tree:

The first three are summariezed as **linear trees**

Join selectivity

- input
  - cardinalities $|R_i|$
  - selectivities $f_{i,j}$: if $p_{i,j}$ is the join predicate between R_i and R_j, define
$$f_{i,j} = \frac{R_i \bowtie_{p_{i,j}} R_j}{R_i \times R_j}$$
- Calculate: $\left|R_i \bowtie_{p_{i,j}} R_j\right| = f_{i,j}|R_i||R_j|$
- Rational: The selectivity can be computed/estimated easily (ideally)

Given a join tree T, the result cardinality $|T|$ can be computed recursively as

$$|T| = \begin{cases} |R_i| & \text{if } T \text{ is a leaf } R_i \\ (\prod_{R_i \in T_1, R_j \in T_2} f_{i,j})|T_1||T_2| & \text{if } T = T_1 \bowtie T_2 \end{cases}$$

assuming independence of the predicates

Given a join tree T, the cost function C_{out} is defined as

$$C_{out}(T) = \begin{cases} 0 & \text{if } T \text{ is a leaf } R_i \\ |T| + C_{out}(T_1) + C_{out}(T_2) \text{if } T = T_1 \bowtie T_2 \end{cases}$$

Consider nested loop join (nlj), hash join (hj), and sort merge join (smj), [?] proposes

$$
\begin{aligned}
C_{nlj}(e_1 \bowtie_p e_2) &= |e_1||e_2| \\
C_{hj}(e_1 \bowtie_p e_2) &= h|e_1| \\
C_{smj}(e_1 \bowtie_p e_2) &= |e_1|\log(|e_1|) + |e_2|\log(|e_2|)
\end{aligned}
$$

where e_i are join trees and h is the average length of the collision chain in the hash table. We will assume $h = 1.2$.

For sequence of join operators $s = s_1 \bowtie \cdots \bowtie s_n$

$$
\begin{aligned}
C_{nlj}(s) &= \sum_{i=2}^{n} |s_1 \bowtie \cdots \bowtie s_{i-1}||s_i| \\
C_{hj}(s) &= \sum_{i=2}^{n} h|s_1 \bowtie \cdots \bowtie s_{i-1}| \\
C_{smj}(s) &= \sum_{i=2}^{n} |s_1 \bowtie \cdots \bowtie s_{i-1}|\log(|s_1 \bowtie \cdots \bowtie s_{i-1}|) + \sum_{i=2}^{n} |s_i|\log(|s_i|)
\end{aligned}
$$

Remark. Note that the aboves cost functions are designed for left-deep trees.

Cost function C_{impl} is **symmetric** if $C_{impl}(e_1 \bowtie^{impl} e_2) = C_{impl}(e_2 \bowtie^{impl} e_1)$

ASI: adjacent sequence interchange

Out basic cost functions can be classified as:

	ASI	¬ASI
symmetric	C_{out}	C_{smj}
¬symmetric	C_{hj}	

1.3.2 Search Space

We distringuish four different dimensions:

1. query graph class: chain, cycle, star, and clique
2. join tree structures: left-deep, zig-zag, or bushy
3. join construction: with or without cross product
4. cost functions: with or without ASI property

In total, 48 different join ordering problems

The number of binary trees with n leave nodes is given by $\mathcal{C}(n-1)$, where $\mathcal{C}(n)$ is defined as

$$\mathcal{C}(n) = \begin{cases} 1 & n = 0 \\ \sum_{k=0}^{n} -1\mathcal{C}(k)\mathcal{C}(n-k-1) & n > 0 \end{cases}$$

It can be written in a closed form as

$$\mathcal{C}(n) = \frac{1}{n+1}\binom{2n}{n}$$

The Catalan numbers grow in the order of $\Theta(4^n/n^{1.5})$

Number of join trees with cross products:

- left deep/right deep: $n!$
- zig-zag: there are $n-1$ join operators, and for every left-deep tree, we can derive zig-zag trees by exchanging the left and right inputs. Hence, from any left-deep tree for n relations, we can derive 2^{n-2} zig-zag trees. Therefore there exists a total of $2^{n-2}n!$ zig-zag trees.
- bushy tree: $n!\mathcal{C}(n-1) = \frac{(2n-2)!}{(n-1)!}$

Chain queries, left-deep join trees, no Cartesian product: let's denote the number of left-deep join trees for a chain query $R_1 - \cdots - R_n$ as $f(n)$. $f(0) = 0$, $f(1) = 1$; for $n > 1$, consider adding R_n to all join trees for $R_1 - \cdots - R_{n-1}$. Let's denote the position of R_{n-1} from the bottom with $k \in [1, n-1]$. Then there are $n-k$ join trees for adding R_n after R_{n-1} and one additional tree if $k = 1$ as R_n can be placed before R_{n-1}. What's more, for R_{n-1} to be k, $R_{n-k} - \cdots - R_{n-2}$ must be below it, which is $f(k-1)$ trees for $n > 1$. Therefore

$$f(n) = 1 + \sum_{k=1}^{n-1} f(k-1) * (n-k) = 2^{n-1}$$

Chain queries, zig-zag join trees, no Cartesian product: $2^{n-2} * 2^{n-1} = 2^{2n-3}$

Chain queries, bushy join trees, no Cartesian product: Every subtree of the join tree must contain a subtrain in order to prevent cross products.

$$f(n) = \begin{cases} 1 & n < 2 \\ \sum_{k=1}^{n-1} 2f(k)f(n-k) & n \geq 2 \end{cases} = 2^{n-1}\mathcal{C}(n-1)$$

Star queries, no Cartesian product: $2 * (n-1)!$ possible left-deep join trees and $2 * (n-1)! * 2^{n-2} = 2^{n-1} * (n-1)!$ zig-zag trees

1.3.3 Greedy Heuristics

Input: a set of relations to be joined and a weight function
Output: a join order S
$S = \epsilon$;
$R = \{R_1, \dots, R_n\}$;
while $!\mathit{empty}(R)$ **do**
  Let k be s.t. $weight(R_k) = \min_{R_i \in R}(weight(R_i))$;
  $R \setminus = R_k$;
  $S \circ = R_k$;
end
Algorithm 1: GreedyJoinOrdering-1($\{R = R_1, \dots, R_n\}$,$w : R \to \mathbb{R}$)

Input: a set of relations to be joined and a weight function
Output: a join order S
$S = \epsilon$;
$R = \{R_1, \dots, R_n\}$;
while $!\mathit{empty}(R)$ **do**
  Let k be s.t. $weight(S, R_k) = \min_{R_i \in R}(weight(S, R_i))$;
  $R \setminus = R_k$;
  $S \circ = R_k$;
end
Algorithm 2: GreedyJoinOrdering-2($\{R = R_1, \dots, R_n\}$,$w : R^* \times R \to \mathbb{R}$)

The above algorithms only generate linear join trees, but Greedy Operator Ordering (GOO) generates bushy join trees.

1.3.4 IKKBZ

The most general case for which a polynomial solution is known is charactized by the following features:

- the query graph must be acyclic
- no cross products are considered
- the search space is restricted to left-deep trees
- the cost function must have the ASI property

Input: a set of relations to be joined and a weight function
Output: a join order S
$S = \epsilon$;
$R = \{R_1, \ldots, R_n\}$;
for $i = 1; i \leq n; ++i$ **do**
  $S = R_i$;
  $R = R \setminus R_i$;
  while $!empty(R)$ **do**
    Let k be s.t. $weight(S, R_k) = \min_{R_i \in R}(weight(S, R_i))$;
    $R \setminus = R_k$;
    $S \circ = R_k$;
  end
  $Solutions + = S$
end
return *cheapest in solutions*

Algorithm 3: GreedyJoinOrdering-3($\{R = R_1, \ldots, R_n\}$,$w : R^* \times R \to \mathbb{R}$)

Input: a set of relations to be joined
Output: join tree
$Trees := \{R_1, \ldots, R_n\}$;
while $|Trees|! = 1$ **do**
  find $T_1, T_j \in Trees$ s.t. $i \neq j$, $\left|T_i \bowtie T_j\right|$ is minimal;
   among all pairs of trees in $Trees$;
  $Trees \setminus = \{T_i, T_j\}$;
  $Trees + = T_i \bowtie T_j$;
end

Algorithm 4: GOO($\{R_1, \ldots, R_n\}$)

The IKKBZ-algorithm considers only join operators that have a cost function of the form

$$cost(R_i \bowtie R_j) = |R_i| * h_j(|R_i|)$$

where each R_j have its own cost function h_j. We denote the set of h_j by H. Let us denote by n_i the cardinality of the relation R_i.

The algorithm works as follows. For every relation R_k it computes the optimal join order under the assumption that R_k is the first relation in the join sequence. The resulting subproblems then resemble a job-scheduling problem.

Given a query graph $G = (V, E)$ and a starting relation R_k, we construct the directed **precedence graph** $G_k^p = (V_k^p, E_k^p)$ rooted in R_k as follows:

1. choose R_k as the root node of G_k^p, $V_k^p = \{R_k\}$
2. while $\left|V_k^p\right| < |V|$, choose $R_i \in V \setminus V_k^p$ s.t. $\exists R_j \in V_k^p : (R_j, R_i) \in E$. Add R_i to V_k^p and $R_j \rightarrow R_i$ to E_k^p

The precedence graph describes the ordering of joins implied by the query graph.

A sequence $S = v_1, \dots, v_k$ of nodes conforms to a precedence graph $G = (V, E)$ if

1. $\forall i \in [2, k] \exists j \in [1, i) : (v_j, v_i) \in E$
2. $\nexists i \in [1, k], j \in (i, k] : (v_j, v_i) \in E$

For non-empty sequence S_1 and S_2 and a precedence graph $G = (V, E)$, we write $S_1 \rightarrow S_2$ if S_1 must occur before S_2, i.e.:

1. S_1 and S_2 conform to G
2. $S_1 \cap S_2 = \emptyset$
3. $\exists v_i, v_j \in V : v_i \in S_1 \wedge v_j \in S_2 \wedge (v_i, v_j) \in E$
4. $\nexists v_i, v_j \in V : v_i \in S_1 \wedge v_j \in V \setminus S_1 \setminus S_2 \wedge (v_i, v_j) \in E$

Further we write

$$\begin{aligned} R_{1,2,\dots,k} &= R_1 \bowtie R_2 \bowtie \cdots \bowtie R_k \\ n_{1,2,\dots,k} &= \left|R_{1,2,\dots,k}\right| \end{aligned}$$

For a given precedence graph, let R_i be a relation and $\mathcal{R}_i$ be the set of relations from which there exists a path to R_i

- in any conforming join tree which includes R_i, all relations from $\mathcal{R}_i$ must be joined first
- all other relations R_j that might be joined before R_i will have no connection to R_i, thus $f_{i,j} = 1$

Hence we can define selectivity of the join with R_i as

$$s_i = \begin{cases} 1 & |\mathcal{R}_i| = 0 \\ \prod_{R_j \in \mathcal{R}_i} f_{i,j} & |\mathcal{R}_i| > 0 \end{cases}$$

If the query graph is a chain, the following conditions holds

$$n_{1,2,\ldots,k+1} = n_{1,2,\ldots,k} * s_{k+1} * n_{k+1}$$

We define $s_1 = 1$. Then we have

$$n_{1,2} = s_2 * (n_1 * n_2) = (s_1 * s_2) * (n_1 * n_2)$$

and, in general,

$$n_{1,2,\ldots,k} = \prod_{i=1}^{k} (s_i * n_i)$$

The costs for a totally ordered precedence graph G can be computed as follows:

$$\begin{aligned} Cost_H(G) &= \sum_{i=2}^{n} [n_{1,2,\ldots,i-1} h_i(n_i)] \\ &= \sum_{i=2}^{n} [(\prod_{j=1}^{i} s_j n_j) h_i(n_i)] \end{aligned}$$

If we choose $h_i(n_i) = s_i n_i$, then $C_H \equiv C_{out}$. If $s_i n_i$ is less than one, we call the join **decreasing** and **increasing** otherwise.

Definition 1.1. Define the cost function C_H as follows

$$\begin{aligned} C_H(\epsilon) &= 0 \\ C_H(R_j) &= 0 \qquad \text{if } R_j \text{ is the root} \\ C_H(R_j) &= h_j(n_j) \qquad \text{else} \\ C_H(S_1 S_2) &= C_H(S_1) + T(S_1) * C_H(S_2) \end{aligned}$$

where

$$T(\epsilon) = 1$$
$$T(S) = \prod_{R_i \in S} (s_i * n_i)$$

By induction, $C_H(G) = Cost_H(G)$

Definition 1.2. Let A and B be two sequences and V and U two non-empty sequences. We say that a cost function C has the **adjacent sequence interchange property** (ASI property) iff there exists a function T and a rank function defined for sequence S as

$$rank(S) = \frac{T(S) - 1}{C(S)}$$

s.t. for non-empty sequences $S = AUVB$ the following holds

$$C(AUVB) \leq C(AVUB) \Leftrightarrow rank(U) \leq rank(V)$$

if $AUVB$ and $AVUB$ satisfy the precedence constraints imposed by a given precedence graph

Lemma 1.3. *C_H has the ASI property*

Definition 1.4. Let $M = \{A_1, \dots, A_n\}$ be a set of node sequences in a given precedence graph. Then M is called a **module** if for all sequences B that do not overlap with the sequences in M one of the following conditions holds:

- $B \rightarrow A_i$, $\forall 1 \leq i \leq n$
- $A_i \rightarrow B$, $\forall 1 \leq i \leq n$
- $B \nrightarrow A_i$ and $A_i \nrightarrow B$, $\forall 1 \leq i \leq n$

Lemma 1.5. *Let C be any cost function with the ASI property and $\{A, B\}$ a module. If $A \rightarrow B$ and additionally $rank(B) \leq rank(A)$, then we can find an optimal sequence among those where B directly follows A*

Proof. Every optimal permutation must have the form (U, A, V, B, W) since $A \rightarrow B$. Assume $V \neq \epsilon$. If $rank(A) \leq rank(V)$, then $rank(B) \leq rank(V)$ and we can exchange V and B. Therefore V is empty. $\square$

If the precedence graph demands $A \to B$ but $rank(B) \leq rank(A)$, we speak of **contradictory sequences** A and B. Since the lemma shows that no non-empty subsequence can occur between A and B, we will combine A and B into a new single node replacing A and B. This node represents a **compound relation** comprising all relations in A and B. Its cardinality is computed by multiplying the cardinalities of all relations in A and B, and its selectivity s is the product of all the selectivities s_i of the relations R_i contained in A and B. The continued process of this step until no more contradictory sequences exits is called **normalization**. The opposite step, replacing a compound node by the sequence of relations it was derived from, is called **denormalization**.

Input: an acyclic query graph G for relations $R_1, \dots, R_n$
Output: the best left-deep tree
$R = \emptyset$;
for $i = 1; i \leq n; ++i$ **do**
    Let G_i be the precedence graph derived from G and rooted at R_i;
    $T =$ `IKKBZ-Sub`(G_i);
    $R = R \cup \{T\}$;
end
return *best of* R

Algorithm 5: `IKKBZ`(G)

Input: a precedence graph G_i for relations $R_1, \dots, R_n$ rooted at some R_i
Output: the optimal left-deep tree under G_i
while G_i *is not a chain* **do**
    let r be the root of a subtree in G_i whose subtrees are chains;
    `IKKBZ-Normalize`(r);
    merge the chains under r according to the rank function in ascending order;
end
`IKKBZ-Denormalize`(G_i);
return G_i

Algorithm 6: `IKKBZ-Sub`(G)

Input: the root r of a subtree T of a precedence graph $G = (V, E)$
Output: a normalized subchain
while $\exists r', c \in V, r \rightarrow^* r', (r', c) \in E : rank(r') > rank(c)$ **do**
  replace r' by a compound relation r'' that represents $r'c$;
end

Algorithm 7: `IKKBZ-Normalize`(r)

1.3.5 The Maximum-Value-Precedence Algorithm

Observations:

- greedy heuristic can produce poor results
- IKKBZ only support acyclic queries and ASI cost functions
- MVP algorithm is a polynomial time heuristic with good results

1.3.6 Dynamic Programming

1.3.7 Simplifying the Query Graph

1.3.8 Adaptive Optimization

1.3.9 Generating Permutations

1.3.10 Transformative Approaches

1.3.11 Randomized Approaches

1.3.12 Metaheuristics

1.3.13 Iterative Dynamic Programming

1.3.14 Order Preserving Joins

1.3.15 Complexity of Join Processing

1.4 Accessing the Data

1.5 Physical Properties

1.6 Query Rewriting

1.7 Self Tuning

2 Transaction System

2.1 Computational Models

2.1.1 Page Model

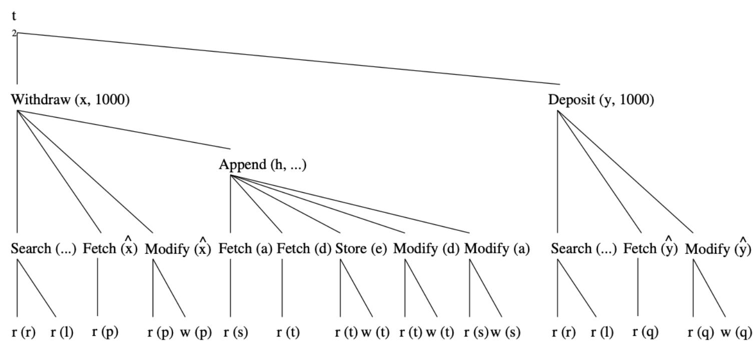
Definition 2.1 (Page Model Transaction)**.** A **transaction** t is a partial order of steps of the form $r(x)$ or $w(x)$ where $x \in D$ and reads and writes as well as multiple writes applied to the same object are ordered. We write $t = (op, <)$ for transaction t with step set op and partial order $<$

2.1.2 Object Model

Definition 2.2 (Object Model Transaction)**.** A **transaction** t is a (finite) tree of labeled nodes with

- the transaction identifier as the label of the root node,

- the names and parameters of invoked operations as labels of inner nodes, and
- page-model read/write operations as labels of leafs nodes, along with a partial order $<$ on the leaf nodes s.t. for all leaf-node operations p and q with p of the form $w(x)$ and q of the form $r(x)$ or $w(x)$ or vice versa, we have $p < q \vee q < p$.



2.2 Notions of Correctness for the Page Model

2.2.1 Canonical Synchronization Problems

Lost Update Problem:

P1	Time	P2
	/* x = 100 */	
r (x)	**1**	
	2	**r (x)**
x := x+100	**4**	**x := x+200**
w (x)	**5**	
	/* x = 200 */	
	6	**w (x)**
	/* x = 300 */	

↑

update “lost”

Observation: problem is the interleaving $r_1(x)\ r_2(x)\ w_1(x)\ w_2(x)$

Inconsistent Read Problem

P1	Time	P2
	1	r (x)
	2	x := x – 10
	3	w (x)
sum := 0	4	
r (x)	5	
r (y)	6	
sum := sum +x	7	
sum := sum + y	8	
	9	r (y)
	10	y := y + 10
	11	w (y)

↑

“sees” wrong sum

Observations:

problem is the interleaving $r_2(x)\ w_2(x)\ r_1(x)\ r_1(y)\ r_2(y)\ w_2(y)$

no problem with sequential execution

Dirty Read Problem

P1	Time	P2
r (x)	1	
x := x + 100	2	
w (x)	3	
	4	r (x)
	5	x := x - 100
failure & rollback	6	
	7	w (x)

↑

cannot rely on validity
of previously read data

Observation: transaction rollbacks could affect concurrent transactions

2.2.2 Syntax of Histories and Schedules

Definition 2.3 (Schedules and histories)**.** Let $T = \{t_1, \ldots, t_n\}$ be a set of transactions, where each $t_i \in T$ has the form $t_i = (op_i, <_i)$

1. A **history** for T is a pair $s = (op(s), <_s)$ s.t.

   (a) $op(s) \subseteq \bigcup_{i=1}^{n} op_i \cup \bigcup_{i=1}^{n} \{a_i, c_i\}$

   (b) for all $1 \leq i \leq n$, $c_i \in op(s) \Leftrightarrow a_i \notin op(s)$

   (c) $\bigcup_{i=1}^{n} <_i \subseteq <_s$

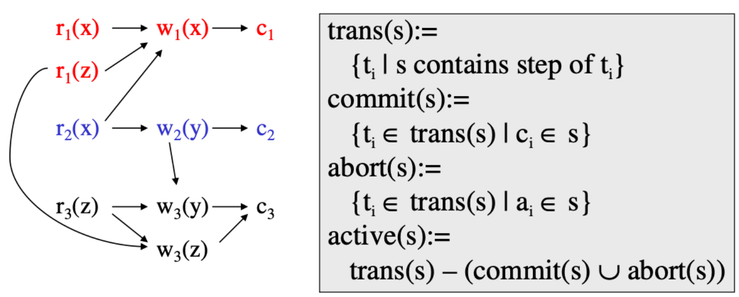
   (d) for all $1 \leq i \leq n$ and all $p \in op_i$, $p <_s c_i \vee p <_s a_i$

(e) for all $p, q \in op(s)$ s.t. at least one of them is a write and both access the same data item: $p <_s q \vee q <_s p$

2. A **schedule** is a prefix of a history

Definition 2.4. A history s is **serial** if for any two transactions t_i and t_j in s, where $i \neq j$, all operations from t_i are ordered in s before all operations from t_j or vice versa

Definition 2.5.
- $trans(s) := \{t_i \mid s \text{ contains step of } t_i\}$
- $commit(s) := \{t_i \in trans(s) \mid c_i \in s\}$
- $abort(s) := \{t_i \in trans(s) \mid a_i \in s\}$
- $active(s) := trans(s) - (commit(s) \cup abort(s))$



$r_1(x)\ r_2(z)\ r_3(x)\ w_2(x)\ w_1(x)\ r_3(y)\ r_1(y)\ w_1(y)\ w_2(z)\ w_3(z)\ c_1\ a_3$

2.2.3 Herbrand Semantics of Schedules

Definition 2.6 (Herbrand Semantics of Steps)**.** For schedule s the **Herbrand semantics** H_s of steps $r_i(x), w_i(x) \in op(s)$ is :

1. $H_s[r_i(x)] := H_s[w_j(x)]$ where $w_j(x)$ is the last write on x in s before $r_i(x)$

2. $H_s[w_i(x)] := f_{ix}(H_x[r_i(y_1)], \ldots, H_s[r_i(y_m)])$ where the $r_i(y_j)$, $1 \leq j \leq m$, are all read operations of t_i that occur in s before $w_i(x)$ and f_{ix} is an uninterpreted m-ary function symbol.

Definition 2.7 (Herbrand Universe)**.** For data items $D = \{x, y, z, \ldots\}$ and transactions t_i, $1 \leq i \leq n$, the **Herbrand universe HU** is the smallest set of symbols s.t.

1. $f_{0x}() \in HU$ for each $x \in D$ where f_{0x} is a constant, and
2. if $w_i(x) \in op_i$ for some t_i, there are m read operations $r_i(y_1), \dots, r_i(y_m)$ that precede $w_i(x)$ in t_i, and $v_1, \dots, v_m \in HU$, then $f_{ix}(v_1, \dots, v_m) \in HU$

Definition 2.8 (Schedule Semantics)**.** The **Herbrand semantics of a schedule** s is the mapping $H[s] : D \to HU$ defined by $H[s](x) := H_s[w_i(x)]$ where $w_i(x)$ is the last operation from s writing x, for each $x \in D$

$$s = w_0(x)\ w_0(y)\ c_0\ r_1(x)\ r_2(y)\ w_2(x)\ w_1(y)\ c_2\ c_1$$

$$\begin{aligned}
&H_s[w_0(x)] = f_{0x}(\,)\\
&H_s[w_0(y)] = f_{0y}(\,)\\
&H_s[r_1(x)] = H_s[w_0(x)] = f_{0x}(\,)\\
&H_s[r_2(y)] = H_s[w_0(y)] = f_{0y}(\,)\\
&H_s[w_2(x)] = f_{2x}(H_s[r_2(y)]) = f_{2x}(f_{0y}(\,))\\
&H_s[w_1(y)] = f_{1y}(H_s[r_1(x)]) = f_{1y}(f_{0x}(\,))
\end{aligned}$$

$$\begin{aligned}
&H[s](x) = H_s[w_2(x)] = f_{2x}(f_{0y}(\,))\\
&H[s](y) = H_s[w_1(y)] = f_{1y}(f_{0x}(\,))
\end{aligned}$$

2.2.4 Final-State Serializability

Definition 2.9. Schedules s and s' are called **final state equivalent**, denoted $s \approx_f s'$ if $op(s) = op(s')$ and $H[s] = H[s']$

Example a:

$$\left.\begin{aligned}
&s = r_1(x)\ r_2(y)\ w_1(y)\ r_3(z)\ w_3(z)\ r_2(x)\ w_2(z)\ w_1(x)\\
&s' = r_3(z)\ w_3(z)\ r_2(y)\ r_2(x)\ w_2(z)\ r_1(x)\ w_1(y)\ w_1(x)\\
&H[s](x) = H_s[w_1(x)] = f_{1x}(f_{0x}(\,)) = H_{s'}[w_1(x)] = H[s'](x)\\
&H[s](y) = H_s[w_1(y)] = f_{1y}(f_{0x}(\,)) = H_{s'}[w_1(y)] = H[s'](y)\\
&H[s](z) = H_s[w_2(z)] = f_{2z}(f_{0x}(\,), f_{0y}(\,)) = H_{s'}[w_2(z)] = H[s'](z)
\end{aligned}\right\} \Rightarrow s \approx_f s'$$

Example b:

$$\left.\begin{aligned}
&s = r_1(x)\ r_2(y)\ w_1(y)\ w_2(y)\\
&s' = r_1(x)\ w_1(y)\ r_2(y)\ w_2(y)\\
&H[s](y) = H_s[w_2(y)] = f_{2y}(f_{0y}(\,))\\
&H[s'](y) = H_{s'}[w_2(y)] = f_{2y}(f_{1y}(f_{0x}(\,)))
\end{aligned}\right\} \Rightarrow \neg\,(s \approx_f s')$$

Definition 2.10 (Reads-from Relation)**.** Given a schedule s, extended with an initial and a final transaction, t_0 and t_∞

1. $r_j(x)$ **reads** x **in** s **from** $w_i(x)$ if $w_i(x)$ is the last write on x s.t. $w_i(x) <_s r_j(x)$

2. The **reads-from relation** of x is

$$RF(s) := \{(t_i, x, t_j) \mid \text{an } r_j(x) \text{ reads } x \text{ from a } w_i(x)\}$$

3. Step p is **directly useful** for step q, denoted $p \to q$, if q reads from p, or p is a read step and q is a subsequent write step of the same transaction. $\to^*$, the **useful relation**, denotes the reflexive and transitive closure of $\to$.

4. Step p is **alive** in s if it is useful for some step from t_∞, i.e.,

$$(\exists q \in t_\infty) p \overset{*}{\to} q$$

and **dead** otherwise

5. The **live-reads-from relation** of s is

$$LRF(s) := \{(t_i, x, t_j) \mid \text{an alive } r_j(x) \text{ reads } x \text{ from } w_i(x)\}$$

Theorem 2.11. *For schedules s and s' the following statements hold:*

1. *$s \approx_f s'$ iff $op(s) = op(s')$ and $LRF(s) = LRF(s')$*

2. *For s let the step graph $D(s) = (V, E)$ be a directed graph with vertices $V := op(s)$ and edges $E := \{(p, q) \mid p \to q\}$, and the reduced step graph $D_1(s)$ be derived from $D(s)$ by removing all vertices that correspond to dead steps. Then $LRF(s) = LRF(s')$ iff $D_1(s) = D_1(s')$*

Proof. For a given schedule s, we can construct a "step graph" $D(s) = (V, E)$ as follows

$$V := op(s)$$
$$E := \{(p, q) \mid p, q \in V, p \to q\}$$

From a step graph $D(s)$, a reduced step graph $D_l(s)$ can be derived by dropping all vertices (and thier incident edges) that represent dead steps. Then the following can be proven:

1. $LRF(s) = LRF(s') \Leftrightarrow D_l(s) = D_l(s')$

   If $D_l(s) \neq D_l(s')$, if there is $r(x) \in D_l(s) \setminus D_l(s')$, then clearly $LRF(s) \neq LRF(s')$; if there is $w_i(x) \in D_l(s) \setminus D_l(s')$, then $(t_i, x, t_\infty) \in LRF(s) \setminus LRF(s')$.

   If $LRF(s) \neq LRF(s')$, suppose $(t_i, x, t_j) \in LRF(s) \setminus LRF(s')$, then clearly $D_l(s) \neq D_l(s')$

2. $s \approx_f s'$ iff $op(s) = op(s')$ and $D_l(s) = D_l(s')$

□

Corollary 2.12. *Final-state equivalence of two schedules s and s' can be decided in time that is polynomial in the length of the two schedules.*

2.2.5 View Serializability

As we have seen, FSR emphasizes steps that are alive in a schedule. However, since the semantics of a schedule and of the transactions occurring in a schedule are unknown, it is reasonable to require that in two equivalent schedules, each transaction reads the same values, independent of its liveliness.

Lost update anomaly: $L = r_1(x)r_2(x)w_1(x)w_2(x)c_1c_2$. History is not FSR, $LRF(L) = \{(t_0, x, t_2), (t_2, x, t_\infty)\}$, $LRF(t_1t_2) = \{(t_0, x, t_1), (t_1, x, t_2), (t_2, x, t_\infty)\}$ and $LRF(t_2t_1) = \{(t_0, x, t_2), (t_2, x, t_1), (t_1, x, t_\infty)\}$

Inconsistent read anomaly: $I = r_2(x)w_2(x)r_1(x)r_1(y)r_2(y)w_2(y)c_1c_2$, history is FSR $LFR(I) = LFR(t_1t_2) = LFR(t_2t_1) = \{(t_0, x, t_2), (t_0, y, t_2), (t_2, x, t_\infty), (t_2, y, t_\infty)\}$

Definition 2.13 (View Equivalence)**.** Schedules s and s' are **view equivalent**, denoted $s \approx_v s'$, if the following hold:

1. $op(s) = op(s')$
2. $H[s] = H[s']$
3. $H_s[p] = H_{s'}[p]$ for all (read or write) steps

Theorem 2.14. *For schedules s and s' the following statements hold.*

1. *$s \approx_v s'$ iff $op(s) = op(s')$ and $RF(s) = RF(s')$*
2. *$s \approx_v s'$ iff $D(s) = D(s')$*

Proof. 1. $\Rightarrow$: Consider a read step $r_i(x)$ from s. Then $H_s[r_i(x)] = H_{s'}[r_i(x)]$ implies that if $r_i(x)$ reads from some step $w_j(x)$ in s, the same holds in s', and vice versa.

$\Leftarrow$: If $RF(s) = RF(s')$, this in particular applies to t_∞; hence $H[s] = H[s']$. Similarly, for all other reads $r_i(x)$ in s, we have $H_s[r_i(x)] = H_{s'}[r_i(x)]$.

Suppose for some $w_i(x)$, $H_s[w_i(x)] \neq H_{s'}[w_i(x)]$. Thus the set of values read by t_i prior to step w_i is different in s and s', a contradiction to our assumption that $RF(s) = RF(s')$.

□

Corollary 2.15. *View equivalence of two schedules s and s' can be decided in time that is polynomial in the length of the two schedules*

Definition 2.16. A schedule s is **view serializable** if there exists a serial schedule s' s.t. $s \approx_v s'$. VSR denotes the class of all view-serializable histories

Theorem 2.17. $VSR \subset FSR$

Theorem 2.18. *Let s be a history without dead steps. Then $s \in VSR$ iff $s \in FSR$*

Theorem 2.19. *The problem of deciding for a given schedule s whether $s \in VSR$ holds is NP-complete*

Definition 2.20 (Monotone Classes of Histories)**.** Let s be a schedule and $T \subseteq trans(s)$. $\pi_T(s)$ denotes the projection of s onto T. A class of histories is called **monotone** if the following holds:

$$\text{If } s \text{ is in } E\text{, then } \Pi_T(s) \text{ is in } E \text{ for each } T \subseteq trans(s)$$

VSR is not monotone

2.2.6 Conflict Serializability

Definition 2.21 (Conflicts and Conflict Relations)**.** Let s be a schedule, $t, t' \in trans(s)$, $t \neq t'$

1. Two data operations $p \in t$ and $q \in t'$ are in **conflict** in s if they access the same data item and at least one of them is a write

2. $conf(s) := \{(p, q) \mid p, q \text{ are in conflict and } p <_s q\}$ is the **conflict relation** of s

Definition 2.22. Schedules s and s' are **conflict equivalent**, denoted $s \approx_c s'$, if $op(s) = op(s')$ and $conf(s) = conf(s')$

Definition 2.23. Schedule s is **conflict serializable** if there is a serial schedule s' s.t. $s \approx_c s'$. CSR denotes the class of all conflict serializable schedules.

Theorem 2.24. $CSR \subset VSR$

Definition 2.25. Let s be a schedule. The **conflict graph** $G(s) = (V, E)$ is a directed graph with vertices $V := commit(s)$ and edges $E := \{(t, t') \mid t \neq t' \wedge \exists p \in t, q \in t' : (p, q) \in conf(s)\}$

Theorem 2.26. *Let s be a schedule. Then $s \in CSR$ iff $G(s)$ is acyclic.*

Proof. $\Rightarrow$: There is a serial history s' s.t. $op(s) = op(s')$ and $conf(s) = conf(s')$. Consider $t, t' \in V$, $t \neq t'$ with $(t, t') \in E$. Then we have

$$(\exists p \in t)(\exists q \in t')p <_s q \wedge (p, q) \in conf(s)$$

Then $p <_{s'} q$. Also all of t occur before all of t' in s'.

Suppose $G(s)$ were cyclic. Then we have a cycle $t_1 \to t_2 \to ... \to t_k \to t_1$. The same cycle also exists in $G(s')$, a contradiction

$\Leftarrow$: $\square$

Corollary 2.27. *Testing if a schedule is in CSR can be done in time polynomial to the schedule's number of transactions*

Commutativity rules:

1. $C_1 : r_i(x)r_j(y) \sim r_j(y)r_i(x)$ if $i \neq j$
2. $C_2 : r_1(x)w_j(y) \sim w_j(y)r_i(x)$ if $i \neq j$ and $x \neq y$
3. $C_3 : w_i(x)w_j(y) \sim w_j(y)w_i(x)$ if $i \neq j$ and $x \neq y$

Ordering rule:

4. C_4: $o_i(x)$, $p_j(y)$ unordered $\Rightarrow o_i(x)p_j(y)$ if $x \neq y$ or both o and p are reads

Definition 2.28. Schedules s and s' s.t. $op(s) = op(s')$ are **commutativity based equivalent**, denoted $s \sim^* s'$, if s can be transformed into s' by applying rules C1, C2, C3, C4 finitely.

Theorem 2.29. *Let s and s' be schedules s.t. $op(s) = op(s')$. Then $s \approx_c s'$ iff $s \sim^* s'$*

Definition 2.30. Schedule s is **commutativity-based reducible** if there is a serial schedule s' s.t. $s \sim^* s'$

Corollary 2.31. *Schedule s is commutativity-based reducible iff $s \in CSR$*

Definition 2.32. Schedule s is **order preserving conflict serializable** if it is conflict equivalent to a serial schedule s' and for all $t, t' \in trans(s)$, if t completely precedes t' in s, then the same holds in s'. OSCR denotes the class of all schedules with this property.

Theorem 2.33. $OCSR \subset CSR$

$s = w_1(x)r_2(x)c_2w_c(y)c_3w_1(y)c_1 \in CSR \setminus OCSR$

Definition 2.34. Schedules s is **commit order preserving conflict serializable** if for all $t_i, t_j \in trans(s)$, if there are $p \in t_i$, $q \in t_j$ with $(p, q) \in conf(s)$, then $c_i <_s c_j$.

COCSR denotes the class of all schedules with this property

Theorem 2.35. $COCSR \subset CSR$

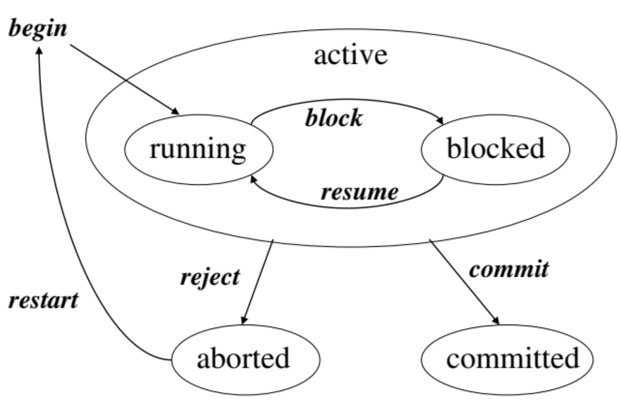
Theorem 2.36. *Schedule s is in COCSR iff there is a serial schedule s' s.t. $s \approx_c s'$ and for all $t_i, t_j \in trans(s)$: $t_i <_{s'} t_j \Leftarrow c_i <_s c_j$*

2.2.7 Commit Serializability

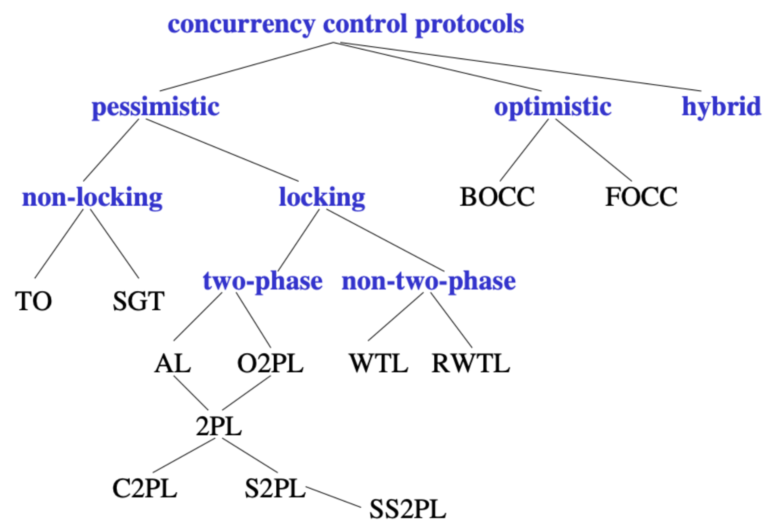
2.2.8 An Alternative Criterion: Interleaving Specifications

2.3 Concurrency Control Algorithms

2.3.1 General Scheduler Design



Definition 2.37 (CSR Safety)**.** For a scheduler S, $Gen(S)$ denotes the set of all schedules that S can generate. A scheduler is called **CSR safe** if $Gen(S) \subseteq CSR$



2.3.2 Locking Schedulers

1. Introduction General locking rules:

   (a) Each data operation $o_i(x)$ must be preceded by $ol_i(x)$ and followed by $ou_i(x)$

   (b) For each x and t_i there is at most one $ol_i(x)$ and at most one $ou_i(x)$

   (c) No $ol_i(x)$ or $ou_i(x)$ is redundant

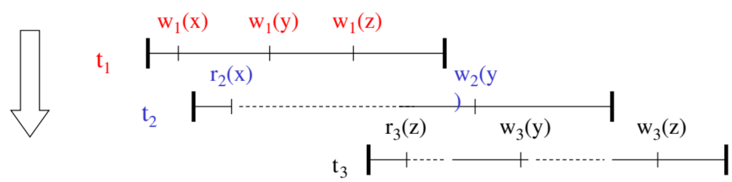
   (d) If x is locked by both t_i and t_j, then these locks are compatible

   Let DT(s) denote the projection of s onto the steps of type r, w, a, c. CP(s) denotes the committed projection of s.

2. Two-Phase Locking

   Definition 2.38. A locking protocol is **two-phase** if for every output schedule s and every transaction $t_i \in trans(s)$ no ql_i step follows the first ou_i step $(q, 0 \in \{r, w\})$

$s = w_1(x)\, r_2(x)\, w_1(y)\, w_1(z)\, r_3(z)\, c_1\, w_2(y)\, w_3(y)\, c_2\, w_3(z)\, c_3$



$wl_1(x)\, w_1(x)\, wl_1(y)\, w_1(y)\, wl_1(z)\, w_1(z)\, wu_1(x)\, rl_2(x)\, r_2(x)\, wu_1(y)\, wu_1(z)\, c_1$
$rl_3(z)\, r_3(z)\, wl_2(y)\, w_2(y)\, wu_2(y)\, ru_2(x)\, c_2$
$wl_3(y)\, w_3(y)\, wl_3(z)\, w_3(z)\, wu_3(z)\, wu_3(y)\, c_3$

Lemma 2.39. *Let s be the output of a 2PL scheduler. Then for each transaction $t_i \in commit(DT(s))$, the following holds:*

(a) *if $o_i(x)$, $o \in \{r, w\}$, occurs in $CP(DT(s))$, then so do $ol_i(x)$ and $ou_i(x)$ with the sequencing $ol_i(x) < o_i(x) < ou_i(x)$.*

(b) *If $t_j \in commit(DT(s))$, $i \neq j$, is another transaction s.t. some steps $p_i(x)$ and $q_j(x)$ from $CP(DT(s))$ are in conflict, then either $pu_i(x) < ql_j$ or $qu_j(x) < pl_i(x)$ holds.*

(c) *If $p_i(x)$ and $q_j(y)$ are in $CP(DT(s))$, then $pl_i(x) < qu_i(y)$, i.e., every lock operation occurs before every unlock operation of the same transaction.*

Lemma 2.40. *Let s be the output of a 2PL scheduler, and let $G := G(CP(DT(s)))$ be the conflict graph of $CP(DT(s))$, then the following holds:*

(a) *If (t_i, t_j) is an edge in G, then $pu_i(x) < ql_j(x)$ for some data item x and two operations $p_i(x)$, $q_j(x)$ in conflict.*

(b) *If $(t_1, \dots, t_n)$ is a path in G, $n \geq 1$, then $pu_1(x) < ql_n(y)$ for two data items x and y as well as operations $p_1(x)$ and $q_n(y)$.*

(c) *G is acyclic.*

Since the conflict graph of an output produced by a 2PL scheduler is acyclic, we have

Theorem 2.41. $Gen(2PL) \subset CSR$

Example 2.1 (Strict inclusion)**.** Let $s = w_1(x)r_2(x)c_2r_3(y)c_3w_1(y)c_1$. $s \in$ CSR as $s \approx_c t_3t_1t_2$. And s cannot be produced by a 2PL scheduler

Theorem 2.42. $Gen(2PL) \subset OCSR$

3. Deadlock Handling Deadlock detection:

   (a) maintain dynamic **waits-for graph** (WFG) with active transactions as nodes and an edge from t_i to t_j if t_j waits for a lock held by t_i

   (b) Test WFG for cycles

   Deadlock resolution: Choose a transaction on a WFG cycles as a **deadlock victim** and abort this transaction, and repeat until no more cycles.

   Possible victim selection strategies:

   (a) Last blocked

   (b) Random

   (c) Youngest

   (d) Minimum locks

   (e) Minimum work

   (f) Most cycles

   (g) Most edges

   Deadlock Prevention: Restrict lock waits to ensure acyclic WFG at all times. Reasonable deadlock prevention strategies when t_i is blocked by t_j:

   (a) **wait-die**: if t_i started before t_j then wait else abort t_i.

   (b) **wound-wait**: if t_i started before t_j then abort t_i else wait

   (c) **Immediate restart**: abort t_i

   (d) **Running priority**: if t_j is itself blocked then abort t_j else wait

   (e) **Timeout**: abort waiting transaction when a timer expires.

   Abort entails later restart

4. Variants of 2PL

   Definition 2.43. Under **static** or **conservative 2PL** (C2PL) each transaction acquires all its locks before the first data operation.

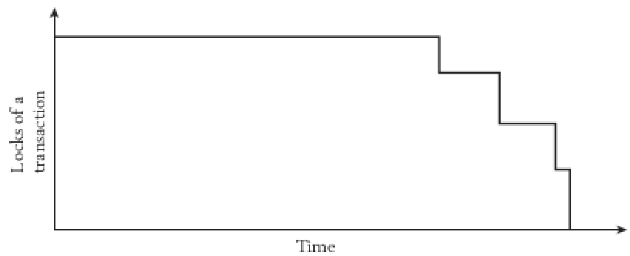


Figure 2: Conservative 2PL

Definition 2.44. Under **strict 2PL** (S2PL) each transaction holds all its write locks until the transaction terminates.

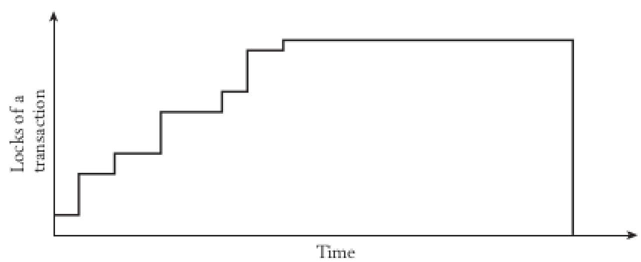


Figure 3: Strict 2PL

Definition 2.45. Under **strong 2PL** (SS2PL) each transaction holds all its locks until the transaction terminates

Theorem 2.46. $Gen(SS2PL) \subset Gen(S2PL) \subset Gen(2PL)$

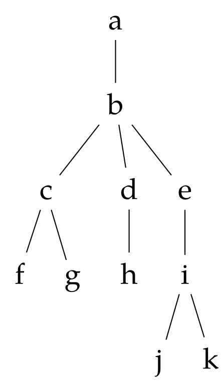
Theorem 2.47. $Gen(SS2PL) \subset COCSR$

5. Ordered Sharing of Locks (O2PL)

6. Altruistic Locking (AL)

7. Non-Two-Phase Locking (WTL, RWTL) Motivation: concurrent executions of transactions with access patterns that comply with organizing data items into a virtual tree

$$t_1 = w_1(a)w_1(b)w_1(d)w_1(e)w_1(i)w_1(k)$$
$$t_2 = w_2(a)w_2(b)w_2(c)w_2(d)w_2(h)$$



Definition 2.48 (Write-only Tree Locking (WTL)). Lock requests and releases must obey LR1 - LR4 and the following additional rules

(a) WTL1: A lock on a node x other than the tree root can be acquired only if the transaction already holds a lock on the parent of x

(b) WTL2: After a $wu_i(x)$ no further $wl_i(x)$ is allowed

8. Geometry of Locking

2.3.3 Non-Locking Schedulers

1. Timestamp Ordering

2.3.4 Hybrid Protocols

2.4 Multiversion Concurrency Control

2.4.1 Multiversion Schedules

Example 2.2. $s = r_1(x)w_1(x)r_2(x)w_2(y)r_1(y)w_1(z)c_1c_2 \notin$ CSR

but schedule would be tolerable if $r_1(y)$ could read the old version y_0 of y to be consistent with $r_1(x)$

Approach:

- each w step creates a new version
- each r step can choose which version it wants/needs to read
- versions are transparent to application and transient

Definition 2.49. Let s be a history with initial transaction t_0 and final transaction t_∞. A **version function** for s is a function h which associates with each read step of s a previous write step on the same data item, and the identity for writes.

Definition 2.50. A **multiversion (mv) history** for transactions $T = \{t_1, \dots, t_n\}$ is a pair $m = (\text{op}(m), <_m)$ where $<_m$ is an order on $\text{op}(m)$ and

1. $\text{op}(m) = \bigcup_{i=1,\dots,n} h(\text{op}(t_i))$ for some version function h
2. for all $t \in T$ and all $p, q \in \text{op}(t_i)$: $p <_t q \Rightarrow h(p) <_m h(q)$
3. if $h(r_j(x)) = w_j(x_i)$, $i \neq j$, then c_i is in m and $c_i <_m c_j$

A **multiversion (mv) schedule** is a prefix of a multiversion history

Definition 2.51. A multiversion schedule is a **monoversion schedule** if its version function maps each read to the last preceding write on the same data item.

2.4.2 Multiversion Serializability

Definition 2.52. For mv schedule m the reads-from relation of m is $\text{RF}(m) = \{(t_i, x, t_j) \mid r_j(x_i) \in \text{op}(m)\}$

Definition 2.53. mv histories m and m' with $trans(m) = trans(m')$ are **view equivalent**, $m \equiv_v m'$, if $\text{RF}(m) = \text{RF}(m')$

2.4.3 Limiting the Number of Versions

2.4.4 Multiversion Concurrency Control Protocols

3 OLAP

3.1 Columar store