Big DataBase

wu

July 4, 2023

Contents

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

1.3.2 Search Space

1.3.3 Greedy Heuristics

1.3.4 IKKBZ

1.3.5 MVP

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

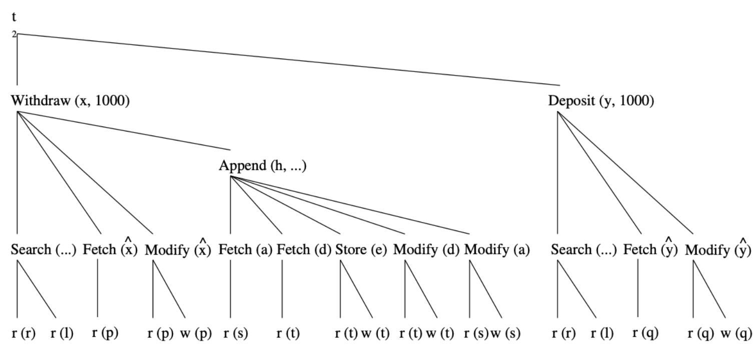
2.1.1 Page Model

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

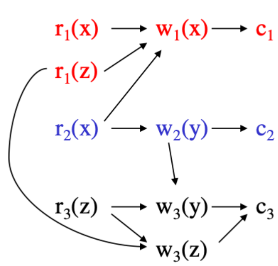
Observation: transaction rollbacks could affect concurrent transactions

2.2.2 Syntax of Histories and Schedules

Definition 2.3 (Schedules and histories)**.** Let $T = \{t_1, \dots, t_n\}$ 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



trans(s):=
  {t_i | s contains step of t_i}
commit(s):=
  {$t_i \in$ trans(s) | $c_i \in$ s}
abort(s):=
  {$t_i \in$ trans(s) | $a_i \in$ 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 2.5 (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. $H_s[w_i(x)] := f_{ix}(H_x[r_i(y_1)], \dots, H_s[r_i(y_m)])$ 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 2.6 (Herbrand Universe)**.** For data items $D = \{x, y, z, \dots\}$ 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 2.7 (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 $x \in D$

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

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

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

2.2.4 Final-State Serializability

Definition 2.8. Schedules s and s' are called **final state equivalent**, denoted $s \approx_f s'$ if $op(s) = op(s')$ and $H[s] = H[s']$

Definition 2.9 (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_∞ 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 2.10. *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 $D_1(s) = D_1(s')$*

Corollary 2.11. *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 2.12 (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 2.13. *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 $RF(s) = RF(s')$.

$\square$

Corollary 2.14. *View equivalence of two schedules s and s' can be decided in time that is polynomial in the length of the two schedules*

Definition 2.15. 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 2.16. $VSR \subset FSR$

Theorem 2.17. *Let s be a history without dead steps. Then $s \in VSR$ iff $s \in FSR$*

Theorem 2.18. *The problem of deciding for a given schedule s whether $s \in VSR$ holds is NP-complete*

Definition 2.19 (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:

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

VSR is not monotone

2.2.6 Conflict Serializability

Definition 2.20 (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. $conf(s) := \{(p, q) \mid p, q$ are in conflict and $p <_s q\}$ is the **conflict relation** of s

Definition 2.21. Schedules s and s' are **conflict equivalent**, denoted $s \approx_c s'$, if $op(s) = op(s')$ and $conf(s) = conf(s')$

Definition 2.22. 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 2.23. $CSR \subset VSR$

Definition 2.24. 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 2.25. *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 2.26. *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 2.27. 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 2.28. *Let s and s' be schedules s.t. $op(s) = op(s')$. Then $s \approx_c s'$ iff $s \sim^* s'$*

Definition 2.29. Schedule s is **commutativity-based reducible** if there is a serial schedule s' s.t. $s \sim^* s'$

Corollary 2.30. *Schedule s is commutativity-based reducible iff $s \in CSR$*

Definition 2.31. 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 2.32. $OCSR \subset CSR$

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

Definition 2.33. 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 2.34. $COCSR \subset CSR$

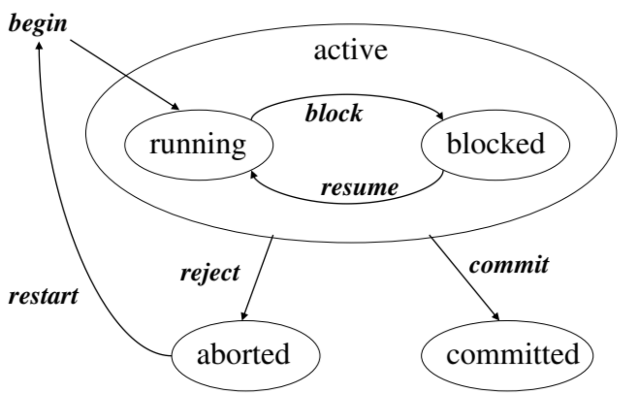
Theorem 2.35. *Schedule s is in COCSR iff there is a serial schedule s' s.t. $s \approx_c s'$ and for all $t_i, t_j \in trans(s)$: $t_i <_{s'} t_j \Leftarrow c_i <_s c_j$*

2.2.7 Commit Serializability

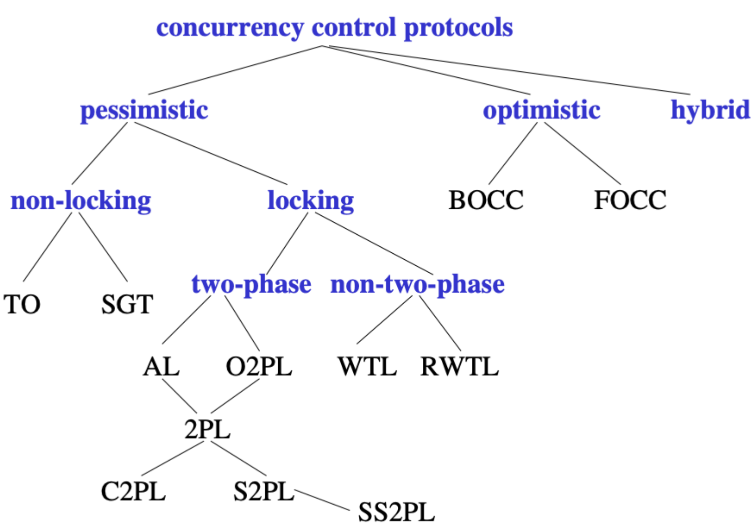
2.2.8 An Alternative Criterion: Interleaving Specifications

2.3 Concurrency Control Algorithms

2.3.1 General Scheduler Design



Definition 2.36 (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

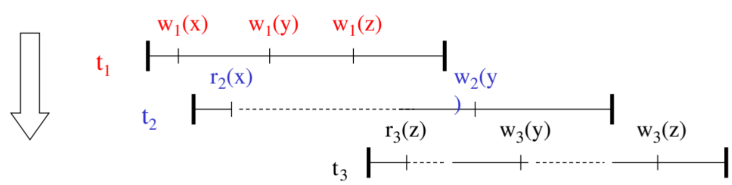
   (d) If x is locked by both t_i and t_j, then these locks are compatible

   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 2.37. 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 2.38. *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 2.39. *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 $(t_1, \ldots, t_n)$ 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 2.40. $Gen(2PL) \subset CSR$

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 2.41. $Gen(2PL) \subset OCSR$

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

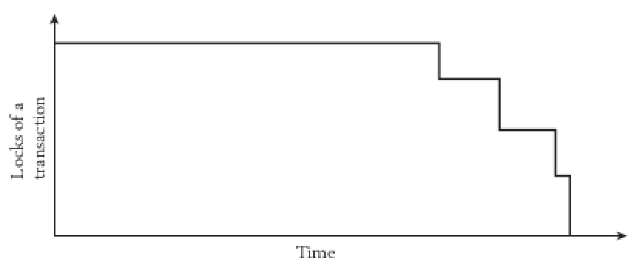


Figure 1: Conservative 2PL

Definition 2.43. Under **strict 2PL** (S2PL) each transaction holds all its write locks until the transaction terminates.

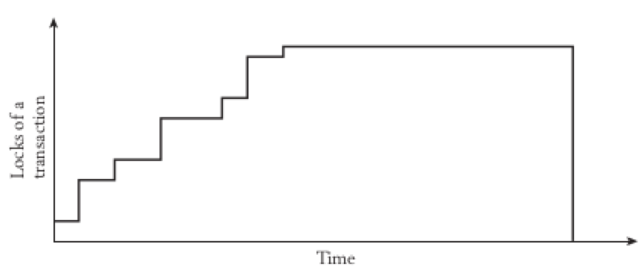


Figure 2: Strict 2PL

Definition 2.44. Under **strong 2PL** (SS2PL) each transaction holds all its locks until the transaction terminates

Theorem 2.45. $Gen(SS2PL) \subset Gen(S2PL) \subset Gen(2PL)$

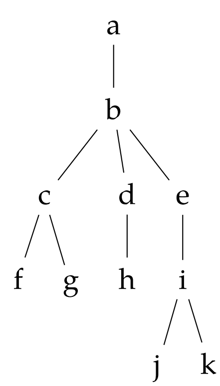
Theorem 2.46. $Gen(SS2PL) \subset COCSR$

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

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



8. Geometry of Locking

2.3.3 Non-Locking Schedulers

2.3.4 Hybrid Protocols