# Big DataBase

wu

May 30, 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, $\{x \circ y \mid x \in e_1 \wedge y \in e_2 \wedge p(x \circ y)\}$

## 1.2 Join Ordering

## 1.3 Accessing the Data

## 1.4 Physical Properties

## 1.5 Query Rewriting

## 1.6 Self Tuning

# 2 Transaction System

## 2.1 Computational Models

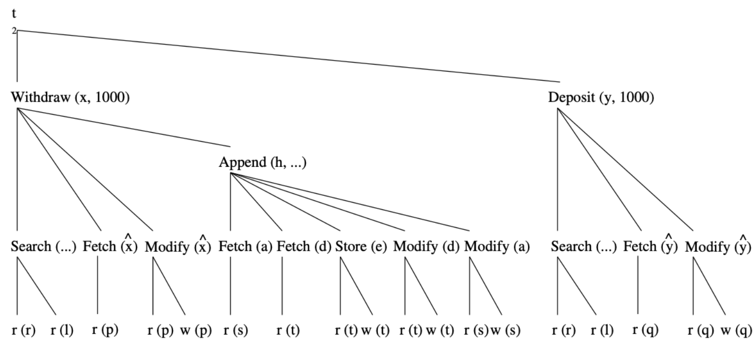
### 2.1.1 Page Model

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

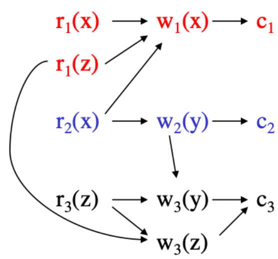
   (c) $\bigcup_{i=1}^{n} <_i \subseteq <_s$

   (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):=
  $\{t_i \mid s$ 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 2.5** (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 hte 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 $x \in D$

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

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

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

### 2.2.4 Final-State Serializability

**Definition 2.8.** 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 $t_\infty$

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 $t_\infty$ 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 $t_\infty$; 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 2.14.** *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 $s$ 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

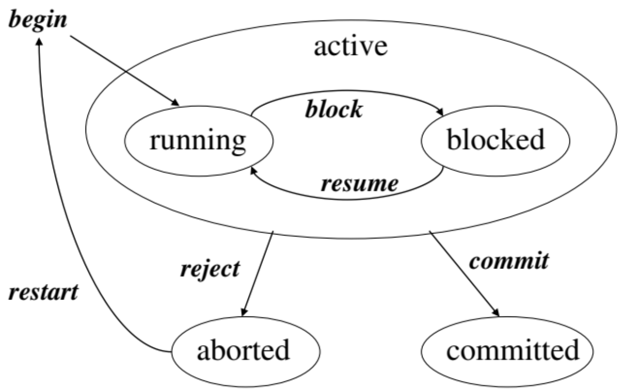
**Theorem 2.28.** $COCSR \subset CSR$

**Theorem 2.29.** *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 An Alternative Criterion: Interleaving Specifications

## 2.3 Concurrency Control Algorithms

### 2.3.1 General Scheduler Design




**Definition 2.30** (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 $Gen(S) \subseteq CSR$

concurrency control protocols
|
pessimistic
|
non-locking
/ \
TO SGT

### 2.3.2 Locking Schedulers

### 2.3.3 Non-Locking Schedulers

### 2.3.4 Hybrid Protocols