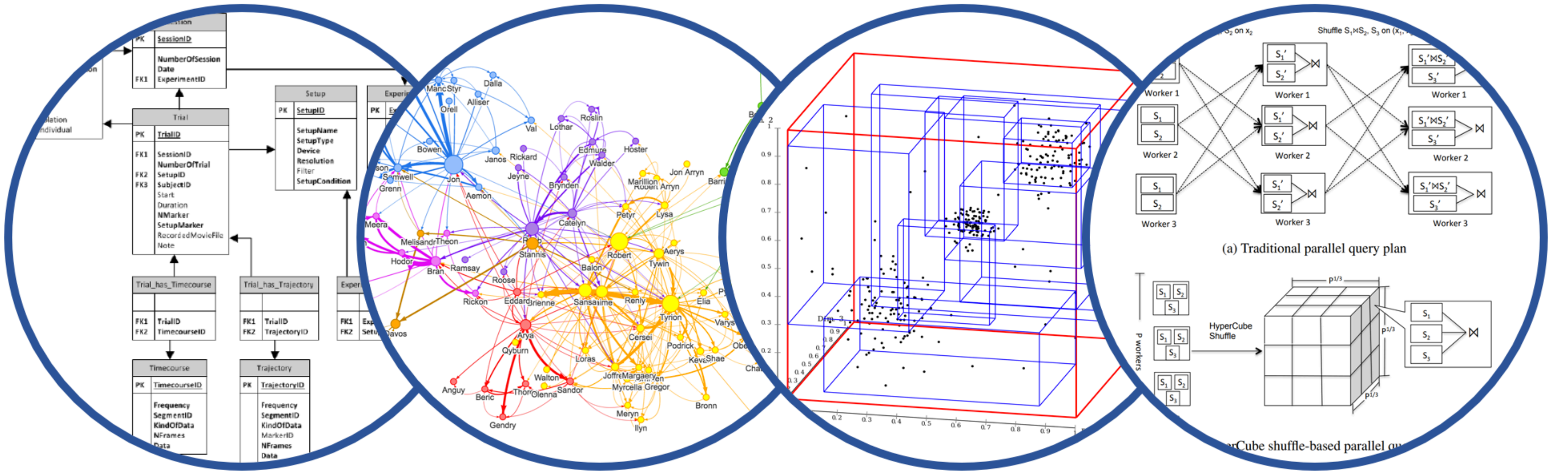


Database System Internals

Concurrency Control Intro

Paul G. Allen School of Computer Science and Engineering
University of Washington, Seattle

Announcements

- Lab2 due on Friday, 5/1/2020

- Quiz: next Wednesday, 5/6/2020
  - Short (30′ or less)
  - *NOT* during the lecture; but a window of time; TDB

- Homework 3 due on Thursday, 5/7/2020

Query Optimization Wrapup

- Cardinality Estimation
- Search Space
- Search Algorithm

Search Algorithm

- Join order using dynamic programming: System R 1979; aka Selinger's algorithm

- Create a table where, for each subquery, we store its size, cost, and optimal plan:
  - {R1}, {R2}, {R3}, ... – single table
  - {R1,R2}, {R1,R3}, ...
  - ...
  - {R1,R2,...,Rn} – entire query

- E.g. entry for {R3,R6,R7,R9}:
  - Consider {R3,R6,R7} $\bowtie$ R9 or {R3,R6,R9} $\bowtie$ R7 or ...
  - Lookup cost/plan for {R3,R6,R7} in table, etc
  - Retain the cheapest cost/plan for {R3,R6,R7,R9}

Search Algorithm – Details

- Runs in exponential time in general
- Heuristics:
  - Left-deep trees only
  - Avoid cartesian products
  - See last few slides of lecture13 for an analysis

Search Algorithm – Details

- "Interesting Orders" is an ordering of a partial result on one of the following:
  - a join attribute, or
  - a group-by attribute, or
  - an order-by attribute

- For each subquery compute *multiple* optimal plans: one for each interesting order

- E.g. $R(A,B) \bowtie S(A,C) \bowtie T(A,D) \bowtie K(D,E)$
  - For {R(A,B),S(A,C)} compute two optimal plans:
    - Unordered, and ordered by A
  - For {R(A,B),T(A,D)} compute three optimal plans:
    - Unordered, ordered by A, ordered by D

Today

- Start discussing transactions
- Lab3 – most difficult lab!
- This lecture and next lecture is mostly review from 344, and we will go quickly
- Later we discuss optimistic concurrency control

Motivating Example

Client 1:
```
UPDATE Budget
SET money=money-100
WHERE pid = 1

UPDATE Budget
SET money=money+60
WHERE pid = 2

UPDATE Budget
SET money=money+40
WHERE pid = 3
```

Client 2:
```
SELECT sum(money)
FROM Budget
```

Would like to treat each group of instructions as a unit

Transaction

Definition: a transaction is a sequence of updates to the database with the property that either all complete, or none completes (all-or-nothing).

```
START TRANSACTION

[SQL statements]

COMMIT    or    ROLLBACK (=ABORT)
```

May be omitted if autocommit is off: first SQL query starts txn

In ad-hoc SQL: each statement = one transaction
This is referred to as autocommit

Motivating Example

```
START TRANSACTION
    UPDATE Budget
    SET money=money-100
    WHERE pid = 1

    UPDATE Budget
    SET money=money+60
    WHERE pid = 2

    UPDATE Budget
    SET money=money+40
    WHERE pid = 3
COMMIT  (or ROLLBACK)
```

```
SELECT sum(money)
FROM Budget
```

With autocommit and without **START TRANSACTION**, each SQL command is a transaction

ROLLBACK

- If the app gets to a place where it can't complete the transaction successfully, it can execute **ROLLBACK**

- This causes the system to "abort" the transaction
  - Database returns to a state without any of the changes made by the transaction

- Several reasons: user, application, system

Transactions

- Major component of database systems
- Critical for most applications; arguably more so than SQL

- Turing awards to database researchers:
  - Charles Bachman 1973
  - Edgar Codd 1981 for inventing relational dbs
  - **Jim Gray 1998 for inventing transactions**
  - Mike Stonebraker 2015 for INGRES and Postgres

ACID Properties

- **A**tomicity: Either all changes performed by transaction occur or none occurs
- **C**onsistency: A transaction as a whole does not violate integrity constraints
- **I**solation: Transactions appear to execute one after the other in sequence
- **D**urability: If a transaction commits, its changes will survive failures

What Could Go Wrong?

Why is it hard to provide ACID properties?

- **Concurrent** operations
  - Isolation problems
  - We saw one example earlier
- **Failures** can occur at any time
  - Atomicity and durability problems
  - Later lectures
- Transaction may need to **abort**

Modeling the Database

- We assume that the database is a set of <u>*elements*</u>

  $X_1, X_2, \ldots, X_n$

- An element can be a record, or a block, or a relation; think of it as being a record

- A transaction performs these actions (any order):
  - reads elements
  - computes
  - writes elements

Schedules

A *schedule* is a sequence of interleaved actions from all transactions

Example

A and B are elements in the database
t and s are variables in tx source code

T1	T2
READ(A, t)	READ(A, s)
t := t+100	s := s*2
WRITE(A, t)	WRITE(A,s)
READ(B, t)	READ(B,s)
t := t+100	s := s*2
WRITE(B,t)	WRITE(B,s)

A Serial Schedule

T1	T2	
		A = 2 B = 2
READ(A, t)		
t := t+100		
WRITE(A, t)		
READ(B, t)		
t := t+100		
WRITE(B,t)		A = 102 B = 102
	READ(A,s)	
	s := s*2	
	WRITE(A,s)	
	READ(B,s)	
	s := s*2	
	WRITE(B,s)	A = 204 B = 204

A Serial Schedule

T1	T2	
		A = 2 B = 2
	READ(A,s)	
	s := s*2	
	WRITE(A,s)	
	READ(B,s)	
	s := s*2	
	WRITE(B,s)	A = 4 B = 4
READ(A, t)		
t := t+100		
WRITE(A, t)		
READ(B, t)		
t := t+100		A = 104 B = 104
WRITE(B,t)		

Serializable Schedule

A schedule is ***serializable*** if it is equivalent to a serial schedule

A Serializable Schedule

T1	T2	A = 2 B = 2
READ(A, t) t := t+100 WRITE(A, t)		A = 102 B = 2
	READ(A,s) s := s*2 WRITE(A,s)	A = 204 B = 2
READ(B, t) t := t+100 WRITE(B,t)		A = 204 B = 102
	READ(B,s) s := s*2 WRITE(B,s)	A = 204 B = 204

This is a serializable schedule.
This is NOT a serial schedule

A Non-Serializable Schedule

T1	T2	
		A = 2 B = 2
READ(A, t) t := t+100 WRITE(A, t)		A = 102 B = 2
	READ(A,s) s := s*2 WRITE(A,s)	A = 204 B = 2
	READ(B,s) s := s*2 WRITE(B,s)	A = 204 B = 4
READ(B, t) t := t+100 WRITE(B,t)		A = 204 B = 104

Serializable Schedules

- The role of the scheduler is to ensure that the schedule is serializable

Q: Why not run only serial schedules ?
I.e. run one transaction after the other ?

Serializable Schedules

- The role of the scheduler is to ensure that the schedule is serializable

Q: Why not run only serial schedules ?
I.e. run one transaction after the other ?

A: Because of very poor throughput due to disk latency.

Lesson: main memory databases *may* schedule TXNs serially

Still Serializable, but…

T1	T2
READ(A, t)	
t := t+100	
WRITE(A, t)	
	READ(A,s)
	s := s + 200
	WRITE(A,s)
	READ(B,s)
	s := s + 200
	WRITE(B,s)
READ(B, t)	
t := t+100	
WRITE(B,t)	

Schedule is serializable because t=t+100 and s=s+200 commute

…we don't expect the scheduler to schedule this

To Be Practical

- Assume worst case updates:
  - Assume cannot commute actions done by transactions
- Therefore, we only care about reads and writes
  - Transaction = sequence of R(A)'s and W(A)'s

T_1: $r_1(A)$; $w_1(A)$; $r_1(B)$; $w_1(B)$
T_2: $r_2(A)$; $w_2(A)$; $r_2(B)$; $w_2(B)$

Conflicts

- Write-Read – WR
- Read-Write – RW
- Write-Write – WW

Conflict Serializability

Conflicts:

Two actions by same transaction T_i: $r_i(X); w_i(Y)$

Two writes by T_i, T_j to same element $w_i(X); w_j(X)$

Read/write by T_i, T_j to same element $w_i(X); r_j(X)$

$r_i(X); w_j(X)$

Conflict Serializability

Definition A schedule is *conflict serializable* if it can be transformed into a serial schedule by a series of swappings of adjacent non-conflicting actions

- Every conflict-serializable schedule is serializable
- The converse is not true in general

Conflict Serializability

Example:

$r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B)$

Conflict Serializability

Example:

$r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B)$

$r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B)$

Conflict Serializability

Example:

$r_1(A); w_1(A); r_2(A); \boxed{w_2(A); r_1(B);} w_1(B); r_2(B); w_2(B)$

$r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B)$

Conflict Serializability

Example:

$r_1(A); w_1(A); r_2(A); \boxed{w_2(A); r_1(B);} w_1(B); r_2(B); w_2(B)$

$r_1(A); w_1(A); r_2(A); r_1(B); w_2(A); w_1(B); r_2(B); w_2(B)$

$r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B)$

Conflict Serializability

Example:

$r_1(A); w_1(A); r_2(A); \boxed{w_2(A); r_1(B);} w_1(B); r_2(B); w_2(B)$

$r_1(A); w_1(A); \boxed{r_2(A); r_1(B);} w_2(A); w_1(B); r_2(B); w_2(B)$

$r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B)$

Conflict Serializability

Example:

$r_1(A); w_1(A); r_2(A); \boxed{w_2(A); r_1(B);} w_1(B); r_2(B); w_2(B)$

$r_1(A); w_1(A); \boxed{r_2(A); r_1(B);} w_2(A); w_1(B); r_2(B); w_2(B)$

$r_1(A); w_1(A); r_1(B); r_2(A); \boxed{w_2(A); w_1(B)}; r_2(B); w_2(B)$

….

$r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B)$

Testing for Conflict-Serializability

Precedence graph:

- A node for each transaction T_i,
- An edge from T_i to T_j whenever an action in T_i conflicts with, and comes before an action in T_j
- No edge for actions in the same transaction

- **The schedule is serializable iff the precedence graph is acyclic**

Example 1

$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$

1 2 3

$r_2(A)$ $r_1(B)$

$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$

1 2 3

$r_2(A)$ $r_1(B)$

$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$

1 2 3

$r_2(A)$ $r_1(B)$

No edge because
no conflict (A != B)

$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$

1 2 3

$\boxed{r_2(A)}$ $\boxed{w_2(A)}$

$\boxed{r_2(A)}$; $r_1(B)$; $\boxed{w_2(A)}$; $r_3(A)$; $w_1(B)$; $w_3(A)$; $r_2(B)$; $w_2(B)$

1 2 3

$r_2(A)$ $w_2(A)$

No edge because
same txn (2)

$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$

1 2 3

$r_2(A)$ $r_3(A)$?

$r_2(A);\ r_1(B);\ w_2(A);\ r_3(A);\ w_1(B);\ w_3(A);\ r_2(B);\ w_2(B)$

1 2 3

$r_2(A)$ $w_1(B)$?

$r_2(A)$; $r_1(B)$; $w_2(A)$; $r_3(A)$; $w_1(B)$; $w_3(A)$; $r_2(B)$; $w_2(B)$

1 2 3

$r_2(A)$ $w_3(A)$?

$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$

1 2 3

$r_2(A)$ $w_3(A)$

Edge! Conflict from T2 to T3

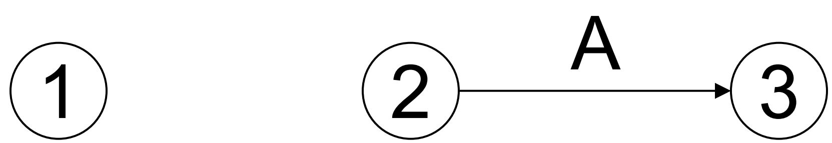
$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$

1 2 3

$\boxed{r_2(A)}$ $\boxed{w_3(A)}$

Edge! Conflict from T2 to T3

$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$



$\boxed{r_2(A)}$ $\boxed{r_2(B)}$?

$\boxed{r_2(A)}$; $r_1(B)$; $w_2(A)$; $r_3(A)$; $w_1(B)$; $w_3(A)$; $\boxed{r_2(B)}$; $w_2(B)$

And so on until compared every pair of actions…

Example 1

$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$

$$1 \xrightarrow{B} 2 \xrightarrow{A} 3$$

More edges, but repeats of the same directed edge not necessary

Example 1

$$r_2(A);\ r_1(B);\ w_2(A);\ r_3(A);\ w_1(B);\ w_3(A);\ r_2(B);\ w_2(B)$$

$$1 \xrightarrow{B} 2 \xrightarrow{A} 3$$

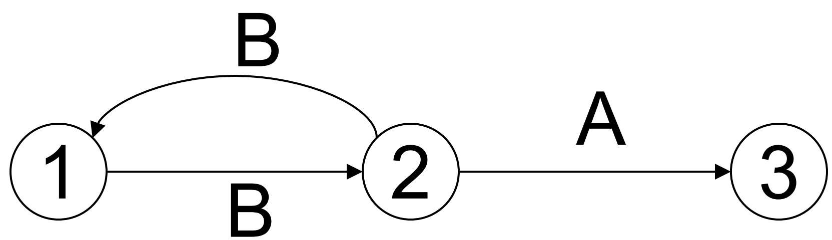
This schedule is **conflict-serializable**

Example 2

$r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B)$

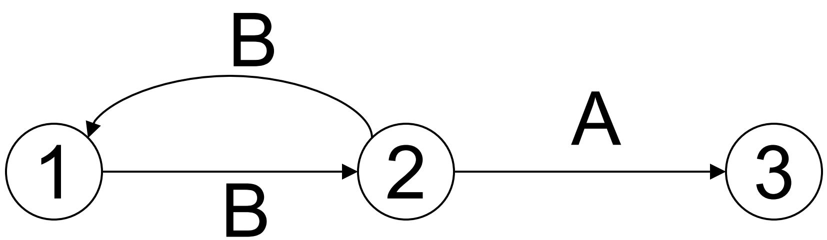
Example 2

$r_2(A);\ r_1(B);\ w_2(A);\ r_2(B);\ r_3(A);\ w_1(B);\ w_3(A);\ w_2(B)$



Example 2

$r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B)$



This schedule **is NOT conflict-serializable**

View Equivalence

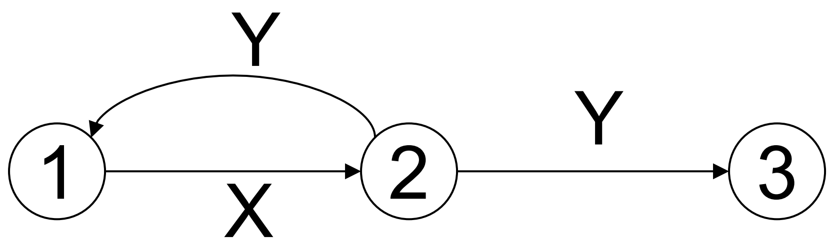
- A serializable schedule need not be conflict serializable, even under the “worst case update” assumption

$$w_1(X);\ w_2(X);\ w_2(Y);\ w_1(Y);\ w_3(Y);$$

Is this schedule conflict-serializable ?

View Equivalence

- A serializable schedule need not be conflict serializable, even under the "worst case update" assumption



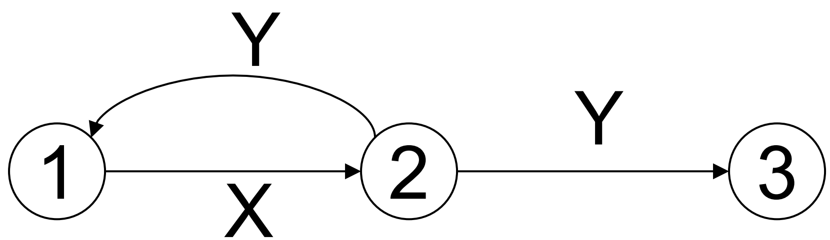
$w_1(X); w_2(X); w_2(Y); w_1(Y); w_3(Y);$

Is this schedule conflict-serializable ?

No…

View Equivalence

- A serializable schedule need not be conflict serializable, even under the “worst case update” assumption



$w_1(X); w_2(X); w_2(Y); w_1(Y); w_3(Y);$

Lost write

$w_1(X); w_1(Y); w_2(X); w_2(Y); w_3(Y);$

Equivalent, but not conflict-equivalent

View Equivalence

T1	T2	T3
W1(X)		
	W2(X)	
	W2(Y)	
	CO2	
W1(Y)		
CO1		
		W3(Y)
		CO3

Lost

T1	T2	T3
W1(X)		
W1(Y)		
CO1		
	W2(X)	
	W2(Y)	
	CO2	
		W3(Y)
		CO3

Serializable, but not conflict serializable

View Equivalence

Two schedules S, S' are *view equivalent* if:

- If T reads an initial value of A in S,
  then T reads the initial value of A in S'

- If T reads a value of A written by T' in S,
  then T reads a value of A written by T' in S'

- If T writes the final value of A in S,
  then T writes the final value of A in S'

View-Serializability

A schedule is *view serializable* if it is view equivalent to a serial schedule

Remark:

- If a schedule is *conflict serializable*, then it is also *view serializable*
- But not vice versa

Schedules with Aborted Transactions

- When a transaction aborts, the recovery manager undoes its updates

- But some of its updates may have affected other transactions !

Schedules with Aborted Transactions

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
	Commit
Abort	

What's wrong?

Schedules with Aborted Transactions

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
	Commit
Abort	

What's wrong?

Cannot abort T1 because cannot undo T2

Recoverable Schedules

A schedule is *recoverable* if:

- It is conflict-serializable, and
- Whenever a transaction T commits, all transactions that have written elements read by T have already committed

Recoverable Schedules

A schedule is *recoverable* if:

- It is conflict-serializable, and
- Whenever a transaction T commits, all transactions that have written elements read by T have already committed

Recoverable Schedules

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
	Commit
?	

Nonrecoverable

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
Commit	
	Commit

Recoverable

Recoverable Schedules

T1	T2	T3	T4
R(A)			
W(A)			
	R(A)		
	W(A)		
	R(B)		
	W(B)		
		R(B)	
		W(B)	
		R(C)	
		W(C)	
			R(C)
			W(C)
			R(D)
			W(D)
Abort			

How do we recover ?

Recoverable Schedules

T1	T2	T3	T4
R(A)			
W(A)			
	R(A)		
	W(A)		
	R(B)		
	W(B)		
		R(B)	
		W(B)	
		R(C)	
		W(C)	
			R(C)
			W(C)
			R(D)
			W(D)
Abort			

Abort

How do we recover ?

Recoverable Schedules

T1	T2	T3	T4
R(A)			
W(A)			
	R(A)		
	W(A)	Abort	
	R(B)		
	W(B)		
		R(B)	
		W(B)	Abort
		R(C)	
		W(C)	
			R(C)
			W(C)
			R(D)
			W(D)
Abort			

How do we recover ?

Recoverable Schedules

T1	T2	T3	T4
R(A)			
W(A)			
	R(A)		
	W(A)		
	R(B)		
	W(B)		
		R(B)	
		W(B)	
		R(C)	
		W(C)	
			R(C)
			W(C)
			R(D)
			W(D)
Abort			

Abort

Abort

Abort

How do we recover ?

Cascading Aborts

- If a transaction T aborts, then we need to abort any other transaction T' that has read an element written by T

- A schedule *avoids cascading aborts* if whenever a transaction reads an element, the transaction that has last written it has already committed.

We base our locking scheme on this rule!

Avoiding Cascading Aborts

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
. . .	
	. . .

With cascading aborts

T1	T2
R(A)	
W(A)	
Commit	
	R(A)
	W(A)
	R(B)
	W(B)
	. . .

Without cascading aborts

Serializability

- Serial
- Serializable
- Conflict serializable
- View serializable

Recoverability

- Recoverable
- Avoids cascading deletes

Scheduler

- The scheduler:
- Module that schedules the transaction's actions, ensuring serializability

- Two main approaches
  - Pessimistic: locks
  - Optimistic: timestamps, multi-version, validation