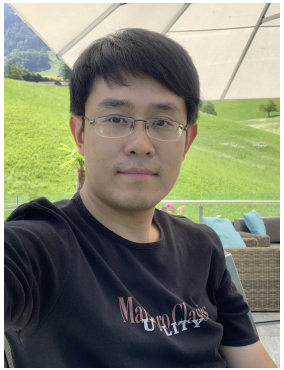
Distributed Snapshots & Global Deadlock Detector

Asim R P
Hubert Zhang
{pasim,zhubert}@vmware.com

Presenters

Asim R P

Pune, India



Hubert Zhang

Beijing, China

Employed by VMware, working on Greenplum database

Outline

Context - sharding using PostgreSQL foreign servers (postgres_fdw)

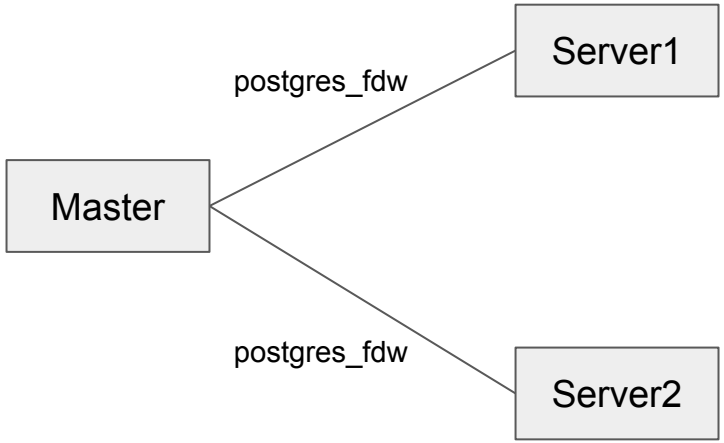
A case of wrong results

Solved with distributed snapshots

Deadlocks go undetected

Solved with global deadlock detection

Distributed setup based on postgres_fdw



Sharding based on FDW

```
create table foo(a int, b varchar) partition by hash(a);

create foreign table foo_s1 partition of foo for values with
(MODULUS 2, REMAINDER 0) SERVER server1 OPTIONS (table_name
'foo');

create foreign table foo_s2 partition of foo for values with
(MODULUS 2, REMAINDER 1) SERVER server2 OPTIONS (table_name
'foo');

insert into foo select i, 'initial insert' from
generate_series(1,100)i;
```

Easy to get wrong results!

```
Transaction1:
        begin isolation level repeatable read;
        insert into foo values (1, 'transaction 1'); -- server1
Transaction2:
        begin isolation level repeatable read;
        insert into foo values (1, 'transaction 2'); -- server1
        insert into foo values (3, 'transaction 2'); -- server2
        commit;
Transaction1:
        select * from foo; -- partial results from transaction2!
```

Demo

What is a snapshot?

```
typedef struct SnapshotData
{
    TransactionId xmin;       /* all XID < xmin are visible to me */
    TransactionId xmax;       /* all XID >= xmax are invisible to me */
    /*
     * note: all ids in xip[] satisfy xmin <= xip[i] < xmax
     */
    TransactionId *xip;
}
```

What is a snapshot?

```
if (tuple.xmin is committed)
{
     if (tuple.xmin <= snapshot.xmin)
          visible
     if (tuple.xmin > snapshot.xmax)
          not visible
     if (tuple.xmin in snapshot.xip[])
          not visible
     ...
}
```

Every tuple is stamped with inserting transaction xid (tuple.xmin)

Snapshot determines if that tuple is visible to current transaction, based on tuple.xmin

Tuples inserted by a transaction that committed before the snapshot was taken are visible

Why did we get wrong results?

server1

	xid	a	b
T1	100	1	‘transaction 1’
T2	101	1	‘transaction 2’

T1 arrives first

T1.xmin = 100

T2 is not visible to T1

server2

	xid	a	b
T2	200	3	‘transaction 2’

T2 arrives first

T1.xmin = 201

T2 is visible to T1

Why did we get wrong results?

T2 is visible to T1’s snapshot server2 but not on server1

(inconsistent snapshots across the cluster)

To get correct results ...

- Global transaction ID service (Postgres-xl)
  - Single point of contention as well as failure
  - Foreign servers cannot be used independently
  -
- Distributed Snapshots
  - Use the same snapshot on all foreign servers
  - Distributed XID assigned by master
  - Tuples record local XID
  - (local XID ←→ distributed XID) mapping on foreign servers
  - Local transactions initiated on foreign servers work as before

Distributed Snapshots

```
XidInMVCCSnapshot()
{
     dxid = distributed_xid(tuple.xmin);
     if (dxid is valid)
          Use distributed snapshot
     else
          Use local snapshot
}
```

Master generates distributed XID and distributed snapshot

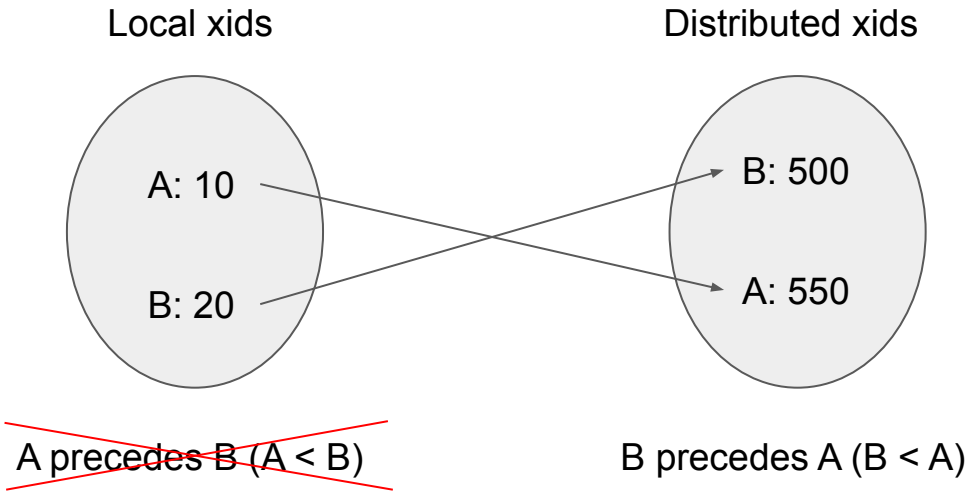
Master sends distributed snapshot along with the query to foreign servers

Local snapshot continues to be created on a foreign server after a query from master arrives

Foreign server keeps a mapping of local to distributed XIDs

Mapping local to distributed xid

- Maintained by each foreign server
- Tuple records local xid
- Distributed xid determines visibility



Distributed Snapshots

server1

	xid	a	b
T1 (dxid 5)	100	1	‘transaction 1’
T2 (dxid 6)	101	1	‘transaction 2’

T1.dxmin < T2.dxmin

T2 is not visible to T1

server2

	xid	a	b
T2 (dxid 6)	200	3	‘transaction 2’

T1 arrives after T2

T1.dxmin < T2.dxmin

T2 is not visible to T1

How long should the mapping last?

- Axioms:
  a. xids are monotonically increasing (local and distributed)
  b. dxid is committed (or aborted) only after local xids on all servers are committed (or aborted)
  c. distributed snapshots arriving at foreign servers are created on the master

- Theorem:

  if dxid is older than the oldest running dxid, its local xid is sufficient to determine visibility

How long should (xid <--> dxid) mapping last?

Distributed snapshot DS: (xmin = 7, xip = [8, 10], xmax = 12)

- The oldest dxid seen as running = 7
- Let dxid = 6 be committed on master (it can no longer be seen as running by axiom a)
- The dxid = 6 is also committed on all foreign servers (axiom b)
- Therefore, on all foreign servers, the local xid for dxid = 6 is also committed
- Let LS: (xmin = 220, xip = …, xmax = …) be the local snapshot on server1 for DS
- Then, local_xid(dxid=6) < 220
- Because local xid for dxid = 6 can no longer be seen as running

Thus, for dxid < 7, local xid is sufficient to determine visibility

Distributed Snapshots

Quick recap:

- Solve wrong results problem with foreign servers
- Created on master, dispatched to servers
- Servers map local xid from a tuple to dxid

Assumption (atomicity):

- When a dxid is committed, its local xids are committed on *all* servers
- Ref: patch “Transactions involving multiple foreign servers”

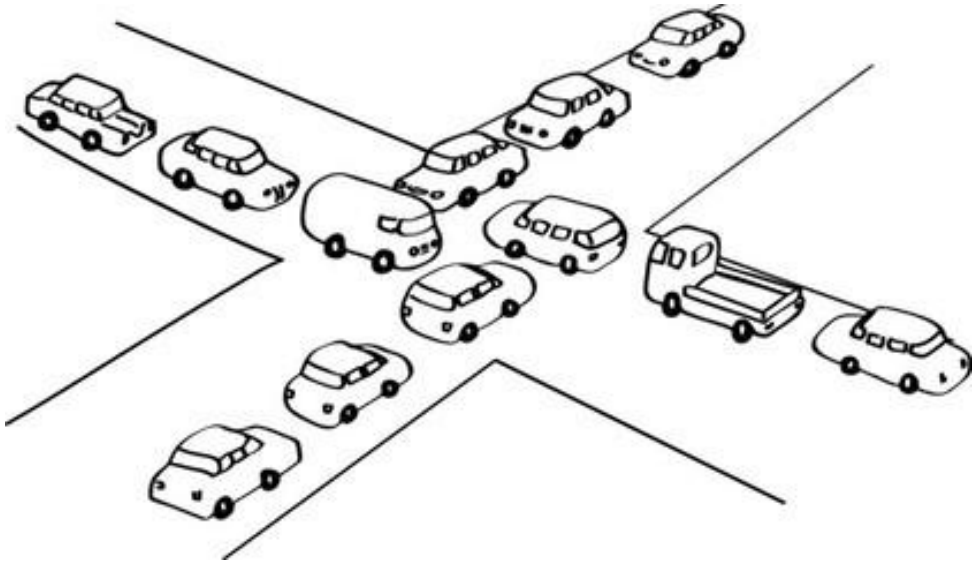
Over to Hubert

Global Deadlock Detector

Deadlock in Single Node

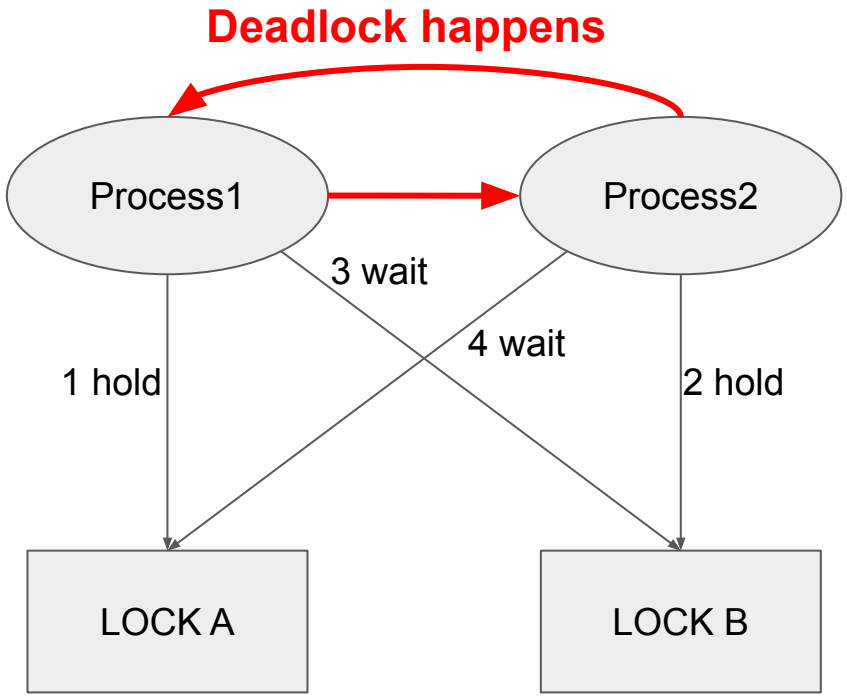
Deadlock in Distributed Cluster

Global Deadlock Detector



https://medium.com/@abhishekdesilva/avoiding-deadlocks-and-performance-tuning-for-mssql-with-wso2-servers-c0014affd1e

Deadlock in Single Node



The **FACT** that process often releases locks at the end of the transaction results in:

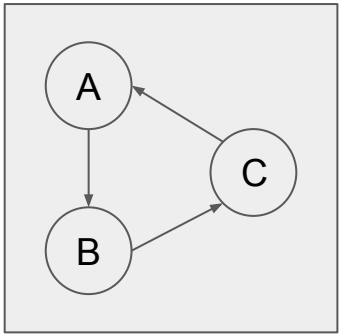
Process1 holds lock A, but waits for lock B.

Process2 holds lock B, but waits for lock A.

Postgres Deadlock Detector

Wait-For Graph

- A graph represents the lock waiting relation among different sessions



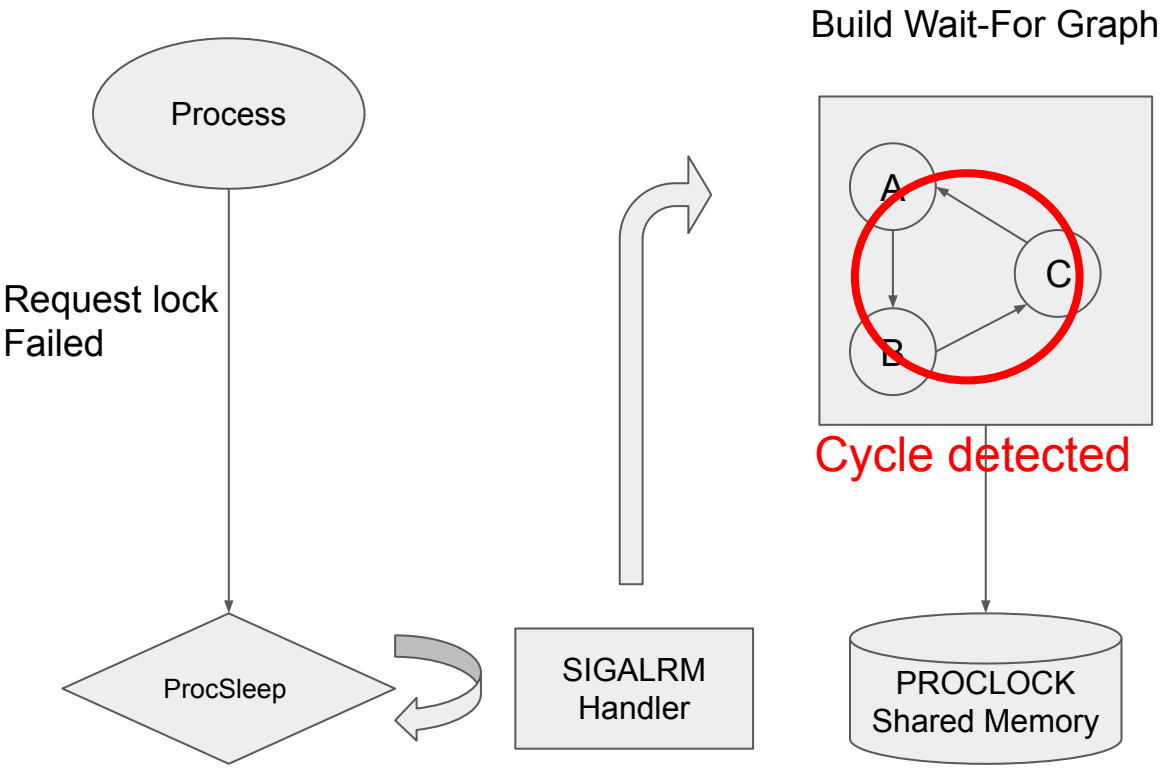
Node

- Process: a postgres backend identifier(pid)

Edge

- Edge represents blocking relationship between processes

Postgres Deadlock Detector

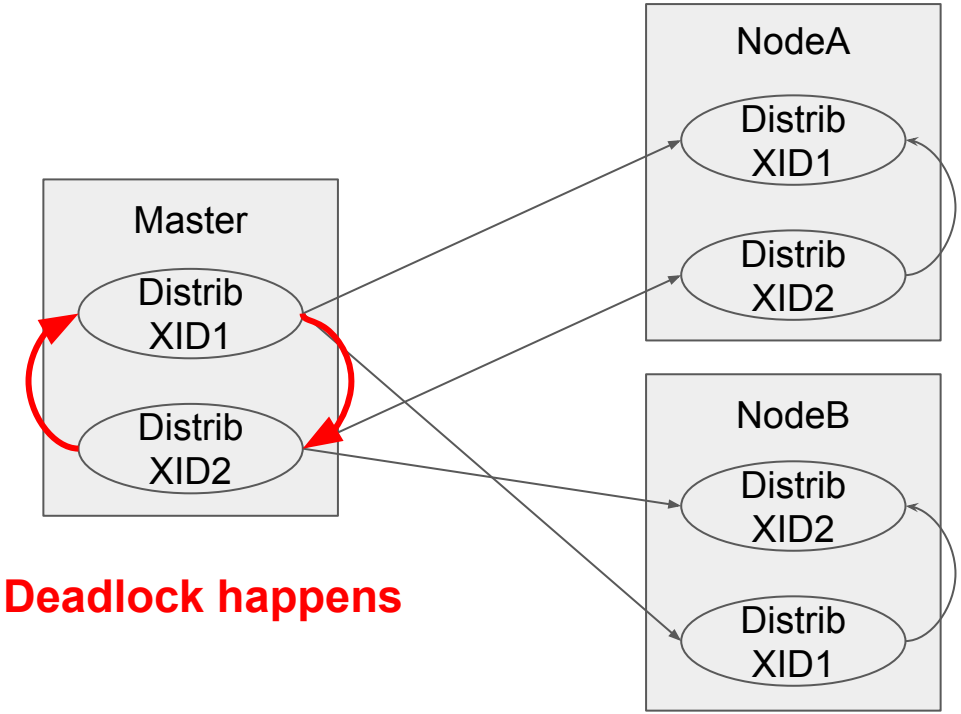


Process will get SIGALRM signal after waiting on a lock for a certain period of time

SIGALRM handler will check shared memory to find the deadlock cycle

Error out the process when cycle detected.

Deadlock in Distributed Cluster



Deadlock happens

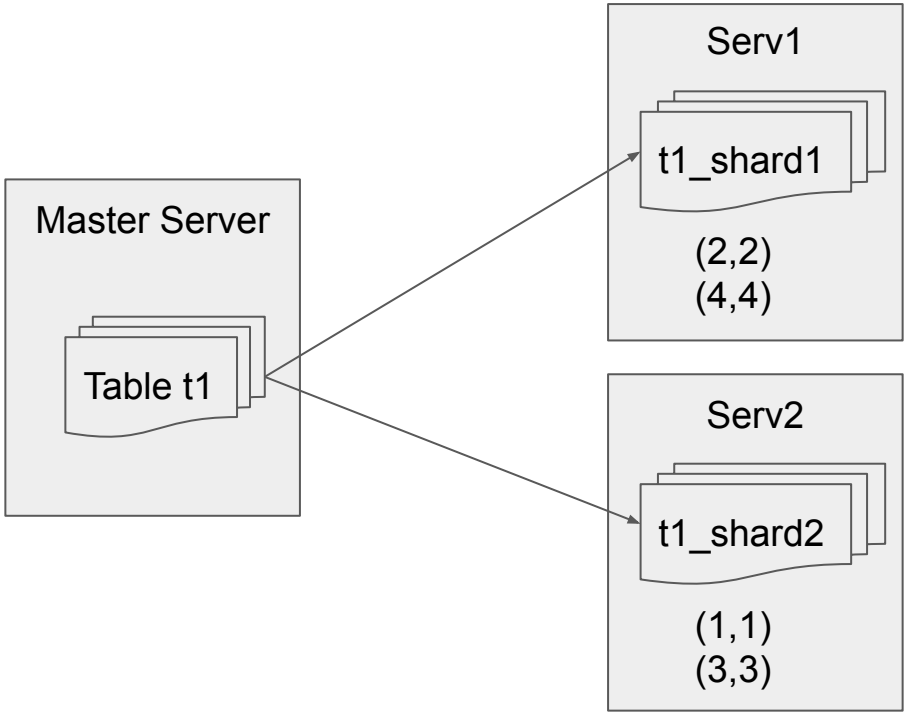
Still the **FACT** that process releases locks at the end of the transaction results in:

Process1 holds lock m on node A, but waits for lock n on node B.

Process2 holds lock n on node B, but waits for lock m on node A.

No deadlock on a local database.

Global Deadlock In FDW cluster




```
CREATE TABLE t1(id int, val int) PARTITION BY
HASH (id);

CREATE FOREIGN TABLE t1_shard1  PARTITION
OF t1  FOR VALUES WITH (MODULUS 2,
REMAINDER 0) SERVER serv1
OPTIONS(table_name 't1');

CREATE FOREIGN TABLE t1_shard2  PARTITION
OF t1 FOR VALUES WITH (MODULUS 2,
REMAINDER 1) SERVER serv2
OPTIONS(table_name 't1');
```

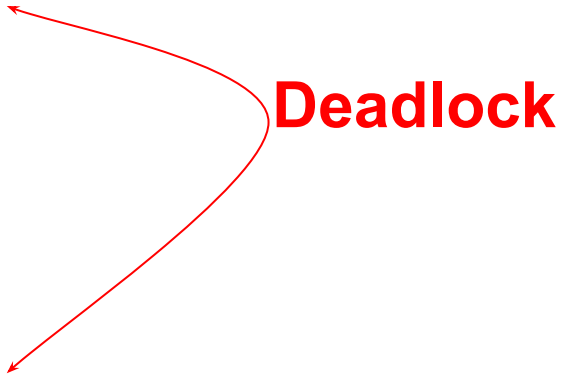
Global Deadlock In FDW cluster

Tx1
huanzhang=# begin;
BEGIN
huanzhang=*# update a set j =3 where **id =1**;
UPDATE 1

huanzhang=*# update a set j =3 where **id =0**;

Tx2
huanzhang=# begin;
BEGIN
huanzhang=*# update a set j =3 where **id=0**;
UPDATE 1

huanzhang=*# update a set j =3 where **id =1**;



Solution

Global Deadlock Detector

Global Deadlock Detector

Postgres Background Worker Based

- Integrate with Postgres ecosystem

Centralized detector

- Single worker process on master to detect deadlock periodically

Full wait-for graph search

- Not effective to find cycle for every vertex.

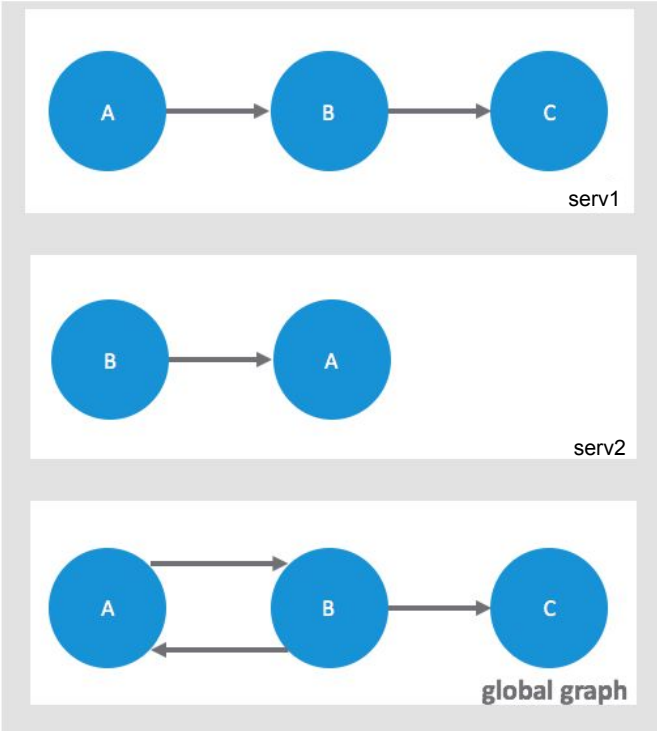
Global Deadlock Detector Component

Wait Graph

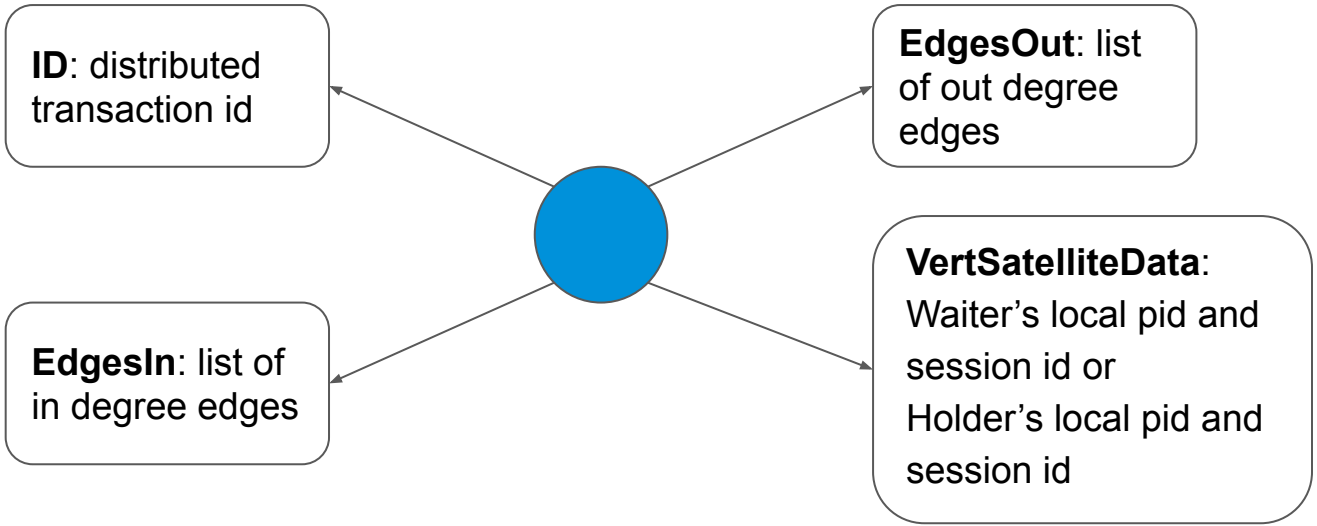
- A graph represents the lock waiting relation **among the database cluster**

Node

- Process group: a session identifier (**distributed transaction id**)



Wait-For Graph Node



Global Deadlock Detector Component

Wait-For Graph

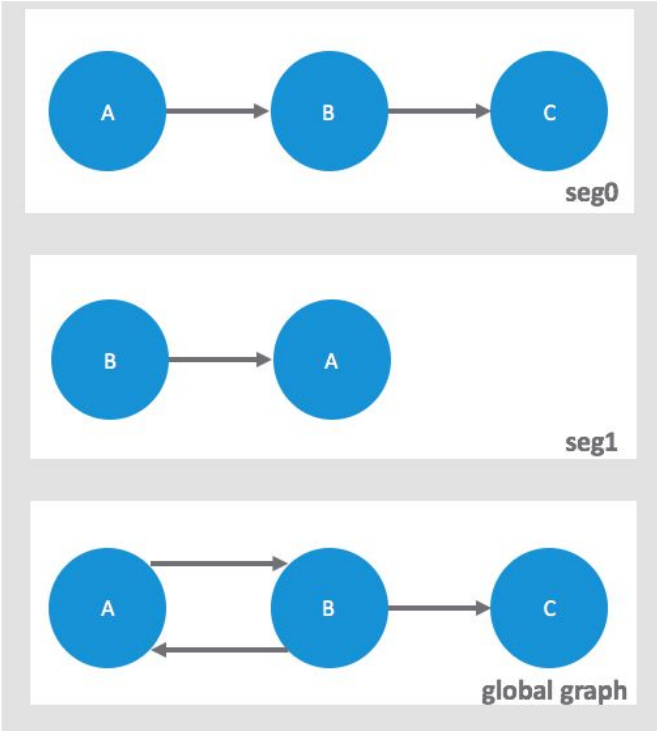
- A graph represents the lock waiting relation **among the database cluster**

Node

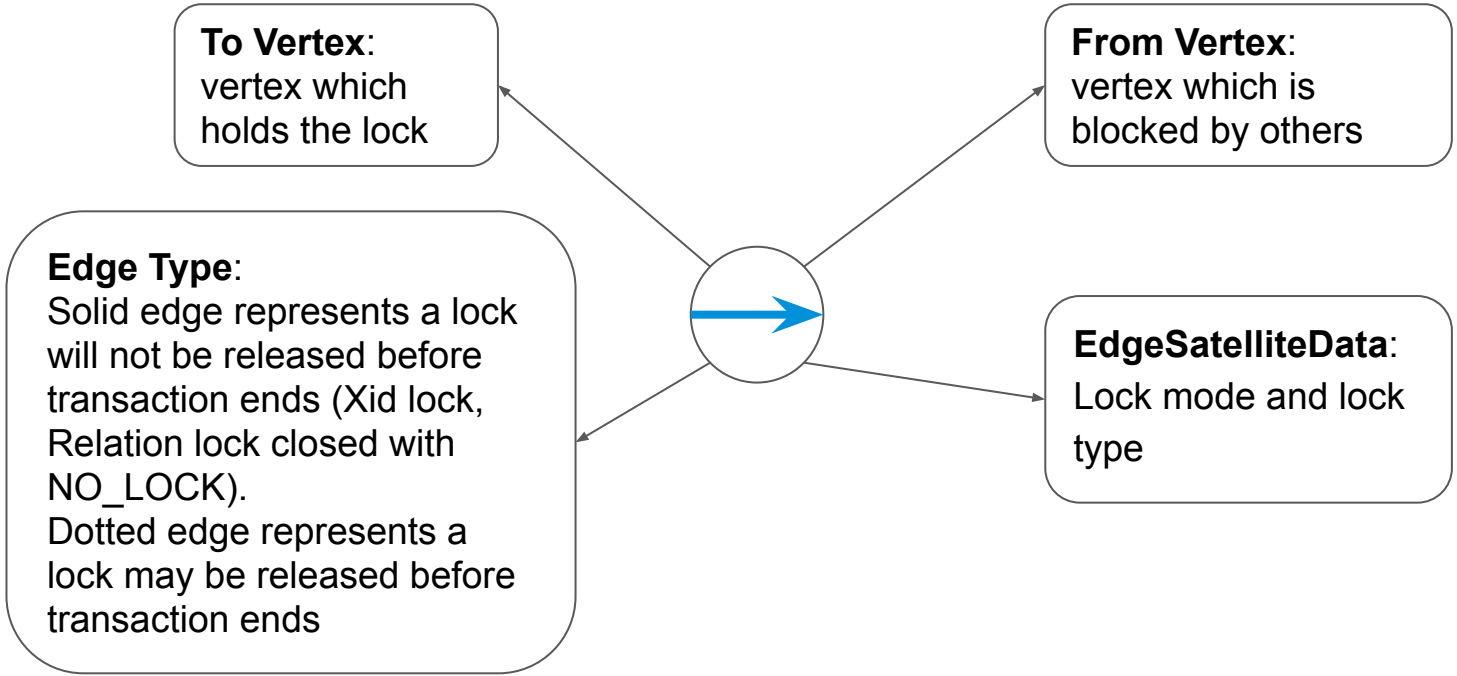
- Process group: a session identifier (**distributed transaction id**)

Edge

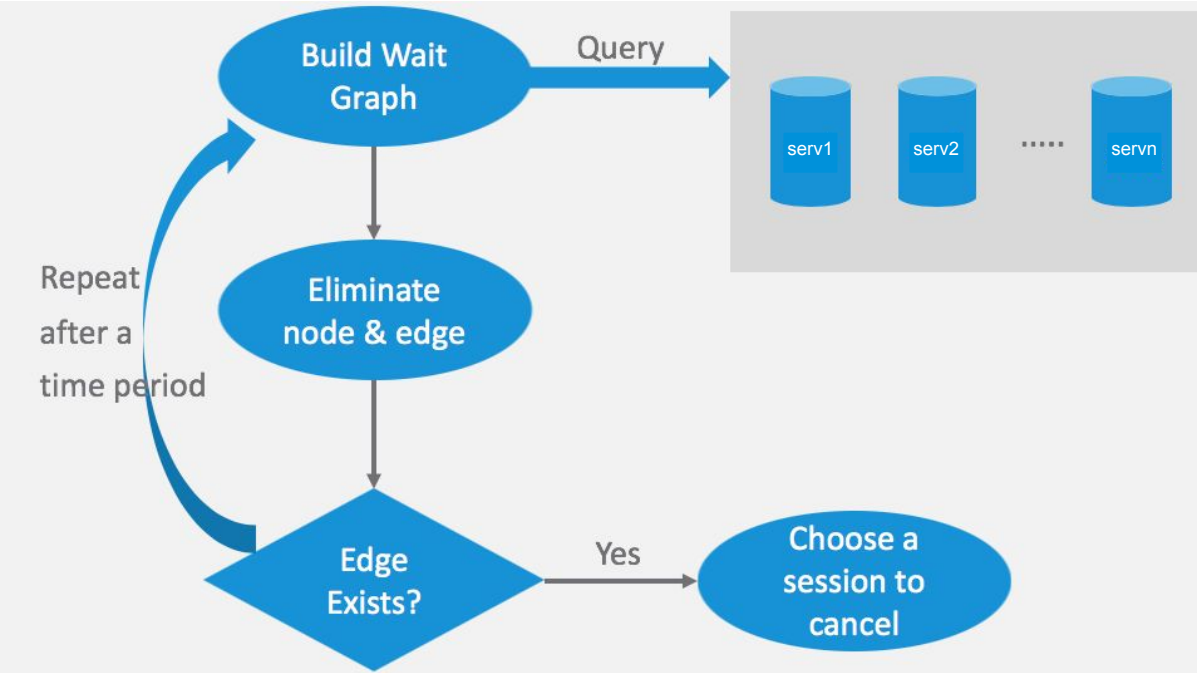
- Edge represents blocking relationship on any one segment



Wait-For Graph Edge



How Would Global Deadlock Detection Work



A dedicate background worker process on master node will build global wait graph periodically by querying the cluster

Node and edge which are not related to deadlock will be eliminated

If edge still exists after eliminating process, report deadlock and cancel a session

Algorithm of Global Deadlock Detector

Build Wait-For Graph

- Gather lock information from shared memory on each segment

Step1: Build Wait-For Graph

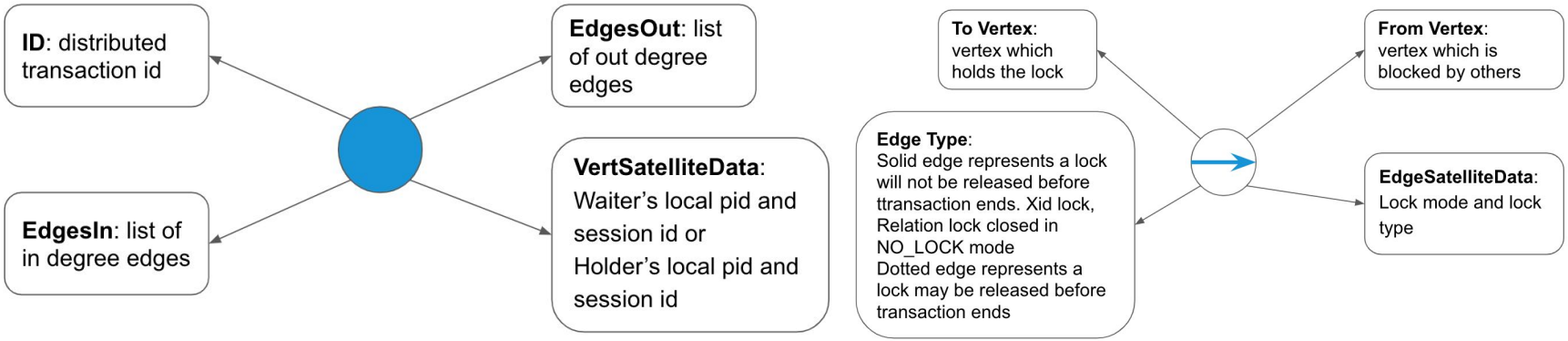
Get Local Wait-For Graph

- Using the Postgres GetLockStatusData function to fetch the lock waiting relationship from PROCLOCK shared memory
- Extending LockInstanceData to include distributed transaction id and holdTillEndXact flag to indicate whether it’s a solid edge or not

Generate Global Wait-For Graph

- Gather result from each foreign servers
- Global graph could be union of edges from all the foreign servers.

Step1: Build Wait-For Graph



servid	waiter_dxid	holder_dxid	holdTillEndXact	waiter_lpid	holder_lpid	Waiter_lckmode	Waiter_lcktype	Waiter_sessionid	Holder_sessionid

Algorithm of Global Deadlock Detector

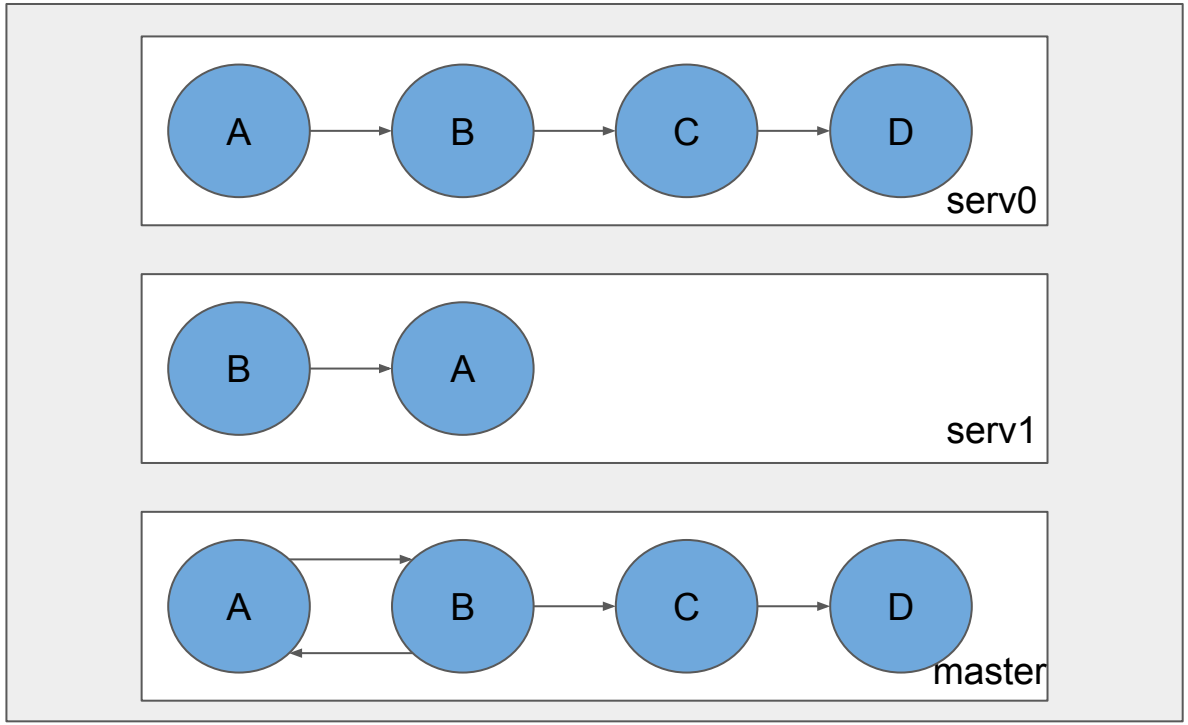
Build Wait-For Graph

- Gather lock information from shared memory in each segments

Find Deadlock

- Greedy algorithm to eliminate node and edge
- Check whether edge still exists in wait-for graph

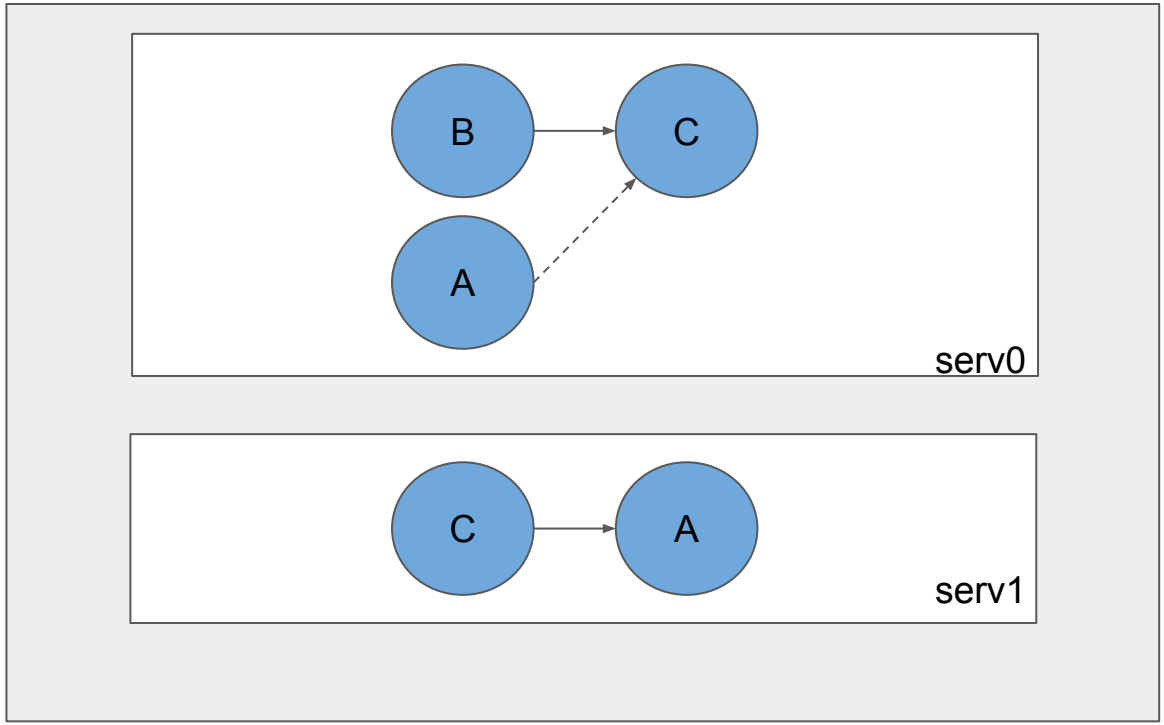
Step2: Eliminating Node & Edge



Greedy on **Global** Wait-For Graph

- Delete all the nodes whose out degree is zero.
- Delete all the corresponding edge point to these node.

Step2: Eliminating Node & Edge



Greedy on **Local** Wait-For Graph

- Find all the dotted edge in each local wait-for graph. If the point to node's out degree is zero, delete this dotted edge.

Algorithm of Global Deadlock Detector

Build Wait-For Graph

- Gather lock information from shared memory in each segments

Find Deadlock

- Greedy algorithm to eliminate node and edge
- Check whether edge still exists in wait-for graph

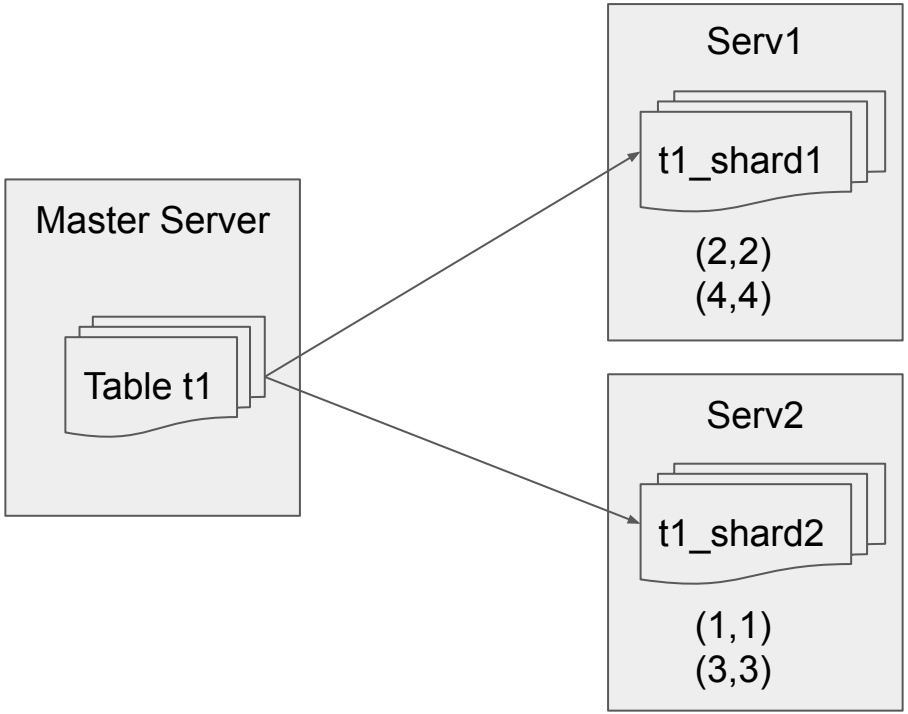
Break Deadlock

- Cancel sessions with strategy: latest session, resource based

Case Study[1]

1 Cases are from “Proposal for distributed deadlock detector”

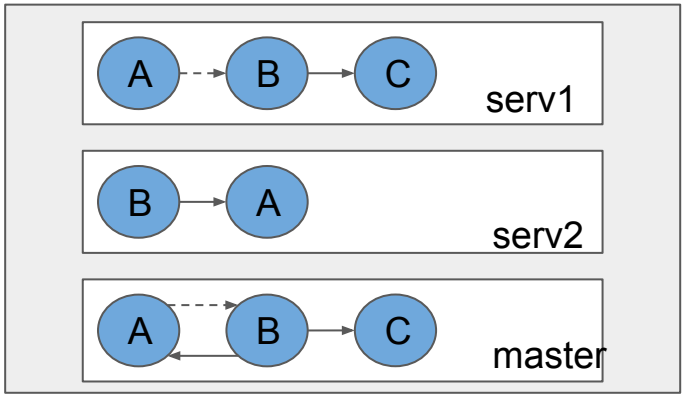
Data Preparation



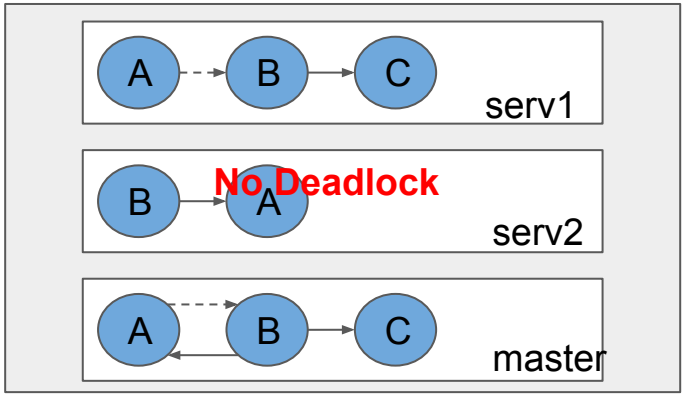

```
CREATE TABLE t1(id int, val int) PARTITION BY
HASH (id);

CREATE FOREIGN TABLE t1_shard1  PARTITION
OF t1  FOR VALUES WITH (MODULUS 2,
REMAINDER 0) SERVER serv1
OPTIONS(table_name 't1');

CREATE FOREIGN TABLE t1_shard2  PARTITION
OF t1 FOR VALUES WITH (MODULUS 2,
REMAINDER 1) SERVER serv2
OPTIONS(table_name 't1');
```

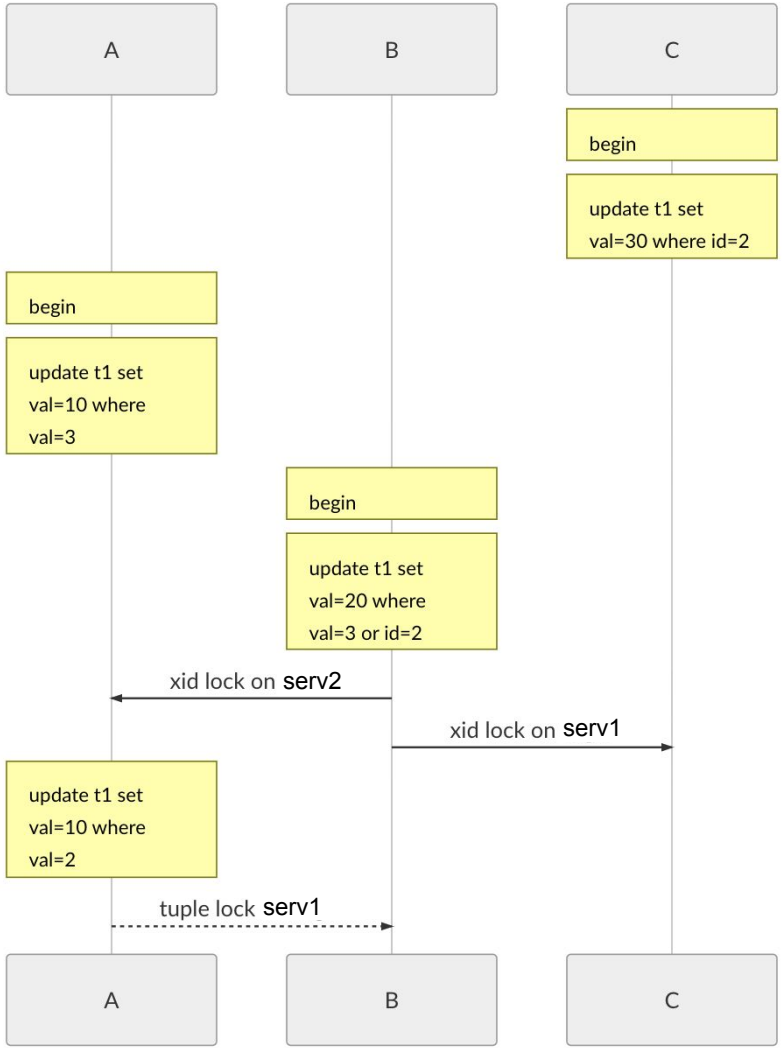
Case 1

Wait-For Graph Before Eliminating



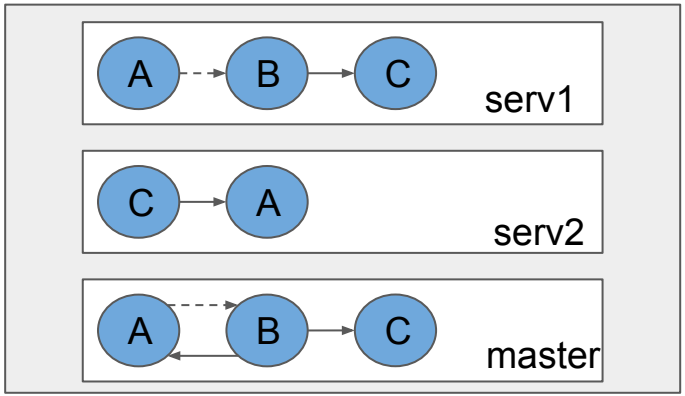
Wait-For Graph After Eliminating



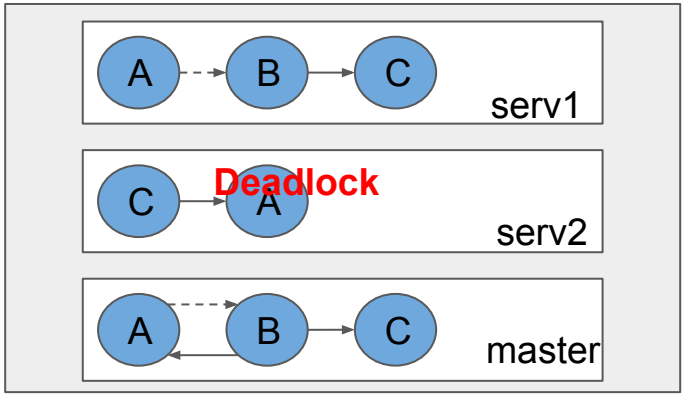


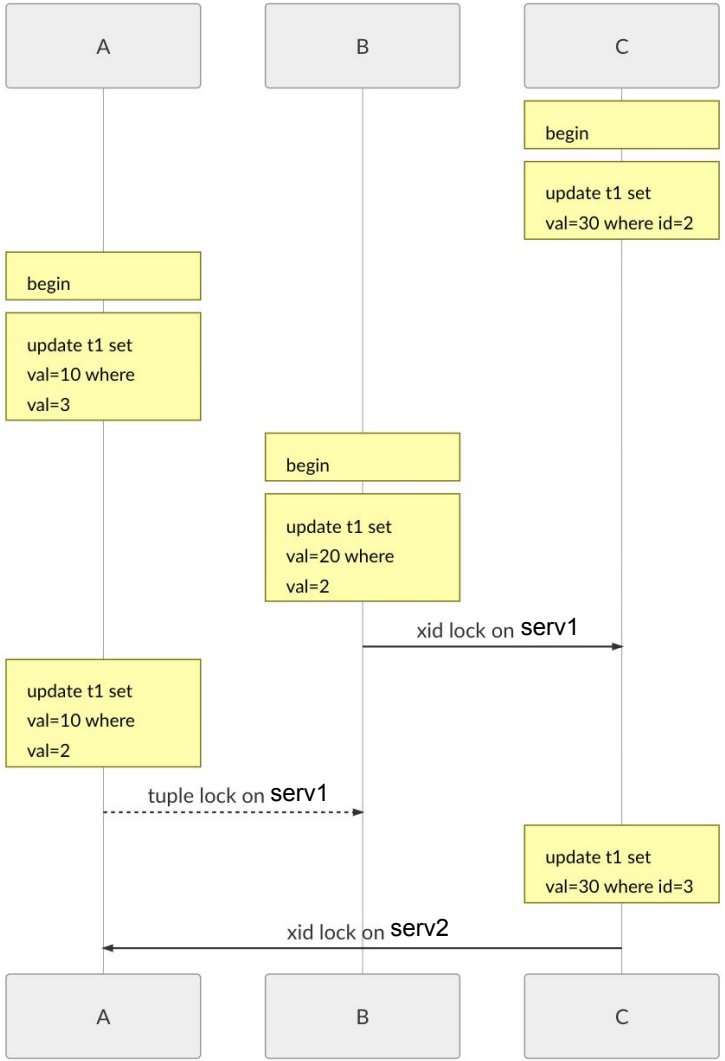
Case 2

Wait-For Graph Before Eliminating



Wait-For Graph After Eliminating






thank you
danke
謝謝
gracias
merci
obrigado
dziękuję
go raibh maith agat
teşekkür ederim
ngiyabonga
спасибо
рахмат
Баярлалаа
bedankt
hvala
mauruuru
köszönöm
sukriya
kop khun krap
terima kasih
감사합니다
ευχαριστώ
arigatô
grazie
dakujem
merci
мерси
dank je
tapadh leat
mochchakkeram
go raibh maith agat
obrigado
sagolun
takk
taiku
xièxie
dankon
akun
gracias
danke
merci