Sharding Using Postgres FDW and Declarative Partitioning

ABOUT SPEAKER

Ajay Reddy Kanduru is a software Engineer with experience in various technologies, including Database Architecture, Python, Django, Celery, and OAuth. I am always curious and eager to learn and explore new advancements in the tech world.

GRIDLEX

AGENDA

- Foreign Data Wrapper
- Declarative Partitioning
- Sharding
- Sharding with postgres_fdw
- Features and Capabilities
- Limitations
- Alternatives for sharding in PostgreSQL
- Conclusion

GRIDLEX

Application Database

1. Orders Table
2. Payments Table
3. Customers Table
4. etc…

Auth Database

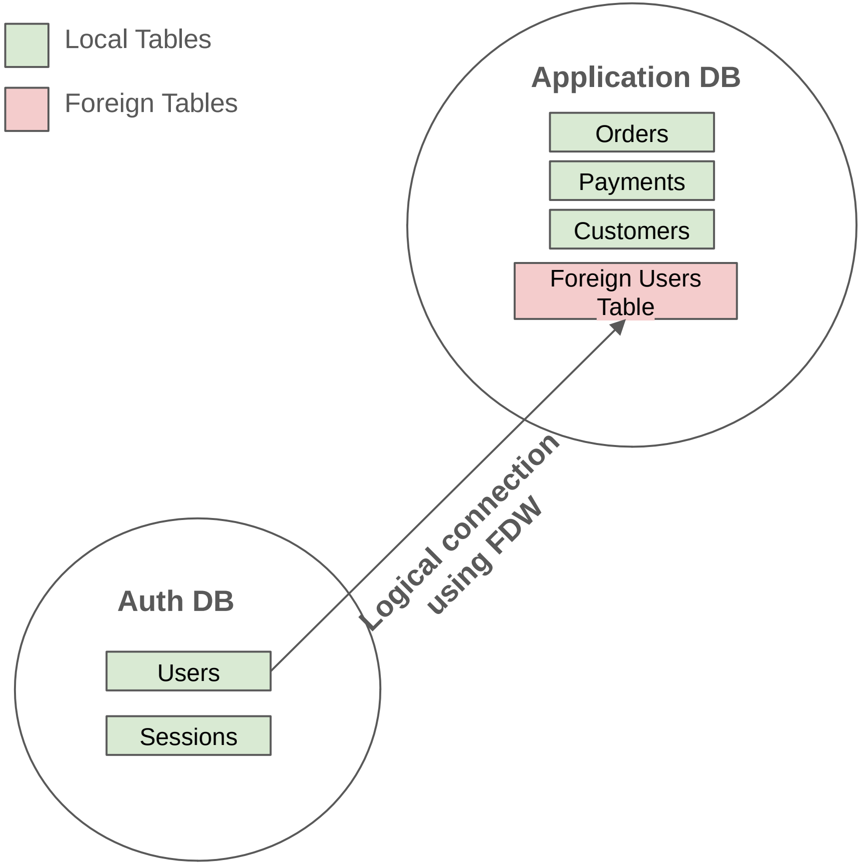
1. Users Table
2. User Session
3. etc…

Display orders along with the user information?

Foreign Data Wrapper

- **Approach 1 :** Fetch the data from two databases separately and combine them at the application level.
- **Approach 2 :** What if you can write join query directly on tables that exist in different databases ?
    - Postgres Foreign Data Wrapper provides this functionality.
    - postgres_fdw is used to connect and access data stored in external Postgresql databases. .

Foreign Data Wrapper with Declarative Partitioning



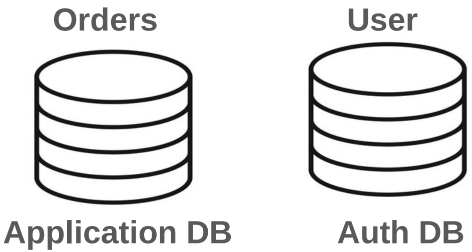

```sql
create extension postgres_fdw;
```

```sql
create server AuthDB
foreign data wrapper postgres_fdw
options (host 'localhost',dbname 'DB2',port '5432');

create user mapping for db1_user server AuthDB
options (user 'db2_user' password 'postgres');
```

```sql
CREATE FOREIGN TABLE foreign_user(
    user_id integer ,
    user_name varchar ,
    country varchar
)
SERVER AuthDB
OPTIONS (table_name 'users');
```

GRIDLEX



On Application DB :

```sql
explain analyse
select * from orders
join foreign_user
on orders.user_id = foreign_user.user_id;
```

```
QUERY PLAN
Hash Join  (cost=101.13..126.00 rows=16 width=64) (actual time=0.737..0.739 rows=0 loops=1)
  Hash Cond: (orders_.user_id = foreign_user.user_id)
  ->  Seq Scan on orders_  (cost=0.00..20.70 rows=1070 width=48) (actual time=0.012..0.015 rows=4 loops=1)
  ->  Hash  (cost=101.09..101.09 rows=3 width=16) (actual time=0.701..0.702 rows=3 loops=1)
        Buckets: 1024  Batches: 1  Memory Usage: 9kB
        ->  Foreign Scan on foreign_user  (cost=100.00..101.09 rows=3 width=16) (actual time=0.685..0.686 rows=3 loops=1)
Planning Time: 0.220 ms
Execution Time: 1.283 ms
```

GRIDLEX

Ecommerce Application

Single Database to store all the data

Lots of Orders are being placed by the Customers
Application is Scaling

GRIDLEX

Problem

Orders Table

Order ID	customer_id	order_date	order amount	Seller ID
101	1001	2024 - 03 - 08	400	10
102	2024	2024 - 03 - 13	590	30
103	2056	2024 - 03 - 28	620	12
104	3124	2024 - 04 - 09	1000	10
105	1001	2024 - 04 - 12	890	54
106	3045	2024 - 04 - 17	654	13

Millions of Orders of all the customers

GRIDLEX

Problem

How to optimise queries on table with millions of rows

GRIDLEX

Partitioning into 10 fragments with hash value

Order ID	Customer id	Order date	order amount	seller_id
101	1000	2024 - 03 - 08	400	201
102	2020	2024 - 03 - 13	590	202
103	3430	2024 - 03 - 28	620	203

Partition 1
Containing data of Customers with hash(customer_id) % 10 = 0

105	981	2024 - 03 - 28	1000	210
164	2231	2024 - 04 - 09	590	212
295	1871	2024 - 04 - 12	850	214

Partition 2
Containing data of Customer with hash(customer_id) % 10 = 1

112	1762	2024 - 04 - 09	400	222
216	2642	2024 - 04 - 12	720	223
326	4742	2024 - 04 - 17	620	250

Partition 3
Containing data of Customer with hash(customer_id) % 10 = 2

Partitioning

Order ID	Customer id	Order date	order amount	seller_id
301	1769	2024 - 03 - 28	400	222
422	2649	2024 - 04 - 09	720	223
453	4749	2024 - 04 - 12	620	250

Partition 10
Containing data of Customer with
hash(customer_id) % 10 = 9

Partitioning

- A table can be divided into multiple smaller fragments with the help of Partitioning.
- **Partitioning key** should be chosen based on which data is distributed.
- Types of Hashing :

Partitioning Strategy	Data Distribution	Sample Business Case
Range Partitioning	Based on consecutive ranges of values	Orders table range partitioned by order_date
List Partitioning	Based on unordered lists of values	Orders table list partitioned by country
Hash Partitioning	Based on a Hash algorithm	Orders table hash partitioned by customer_id

- Here in our case **customer_id** is the partitioning key and we do **Hash based Partitioning** .

Declarative Partitioning

```
CREATE TABLE orders
(
      order_id     INTEGER,
      customer_id INTEGER,
      order_date  DATE,
      order_amount DECIMAL,
      seller_id   INTEGER
)
PARTITION BY hash(customer_id);
```

```
CREATE TABLE orders_part_1 PARTITION OF orders FOR VALUES WITH (modulus 2 , remainder 0);
CREATE TABLE orders_part_2 PARTITION OF orders FOR VALUES WITH (modulus 2 , remainder 1);
```

GRIDLEX

```
insert into orders(order_id, customer_id, order_date, order_amount, seller_id)
values ( 101, 210, '2024-03-08', 400, 13);

insert into orders(order_id, customer_id, order_date, order_amount, seller_id)
values ( 102, 211, '2024-03-24', 450, 14);

insert into orders(order_id, customer_id, order_date, order_amount, seller_id)
 values ( 103, 212, '2024-04-05', 720, 15);

insert into orders(order_id, customer_id, order_date, order_amount, seller_id)
values ( 107, 217, '2024-05-12', 876, 19);
```

Declarative Partitioning

```
select * from orders_part_1;
```

order_id	customer_id	order_date	order_amount	seller_id
101	210	2024-03-08	400	13
103	212	2024-04-05	720	15

```
select * from orders_part_2;
```

order_id	customer_id	order_date	order_amount	seller_id
102	211	2024-03-24	450	14
107	217	2024-05-12	876	19

```
select * from orders;
```

order_id	customer_id	order_date	order_amount	seller_id
101	210	2024-03-08	400	13
103	212	2024-04-05	720	15
102	211	2024-03-24	450	14
107	217	2024-05-12	876	19

Declarative Partitioning

```sql
explain analyse
select * from orders;
```

QUERY PLAN
Append (cost=0.00..52.10 rows=2140 width=48) (actual time=0.014..0.022 rows=4 loops=1)
-> Seq Scan on orders_part_1 orders_1 (cost=0.00..20.70 rows=1070 width=48) (actual time=0.013..0.015 rows=2 loops=1)
-> Seq Scan on orders_part_2 orders_2 (cost=0.00..20.70 rows=1070 width=48) (actual time=0.003..0.004 rows=2 loops=1)
Planning Time: 0.092 ms
Execution Time: 0.044 ms

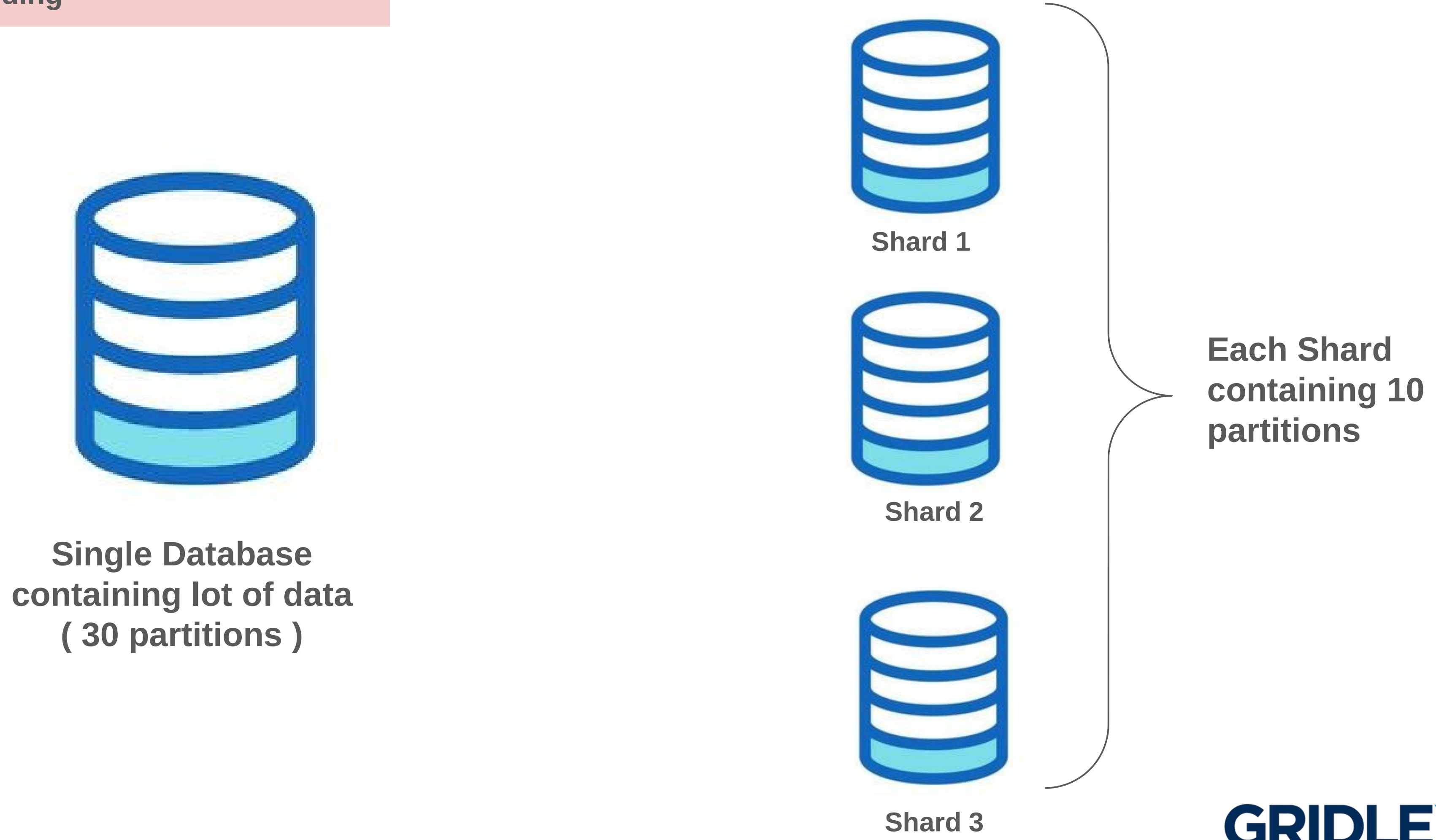
```sql
explain analyse
select * from orders where customer_id=210;
```

QUERY PLAN
Seq Scan on orders_part_1 orders (cost=0.00..23.38 rows=5 width=48) (actual time=0.015..0.018 rows=1 loops=1)
Filter: (customer_id = 210)
Rows Removed by Filter: 1
Planning Time: 0.103 ms
Execution Time: 0.036 ms

GRIDLEX

Even after partitioning the table, we still end up with lot of
data and the server is running out of resources
What can we do ?

Sharding



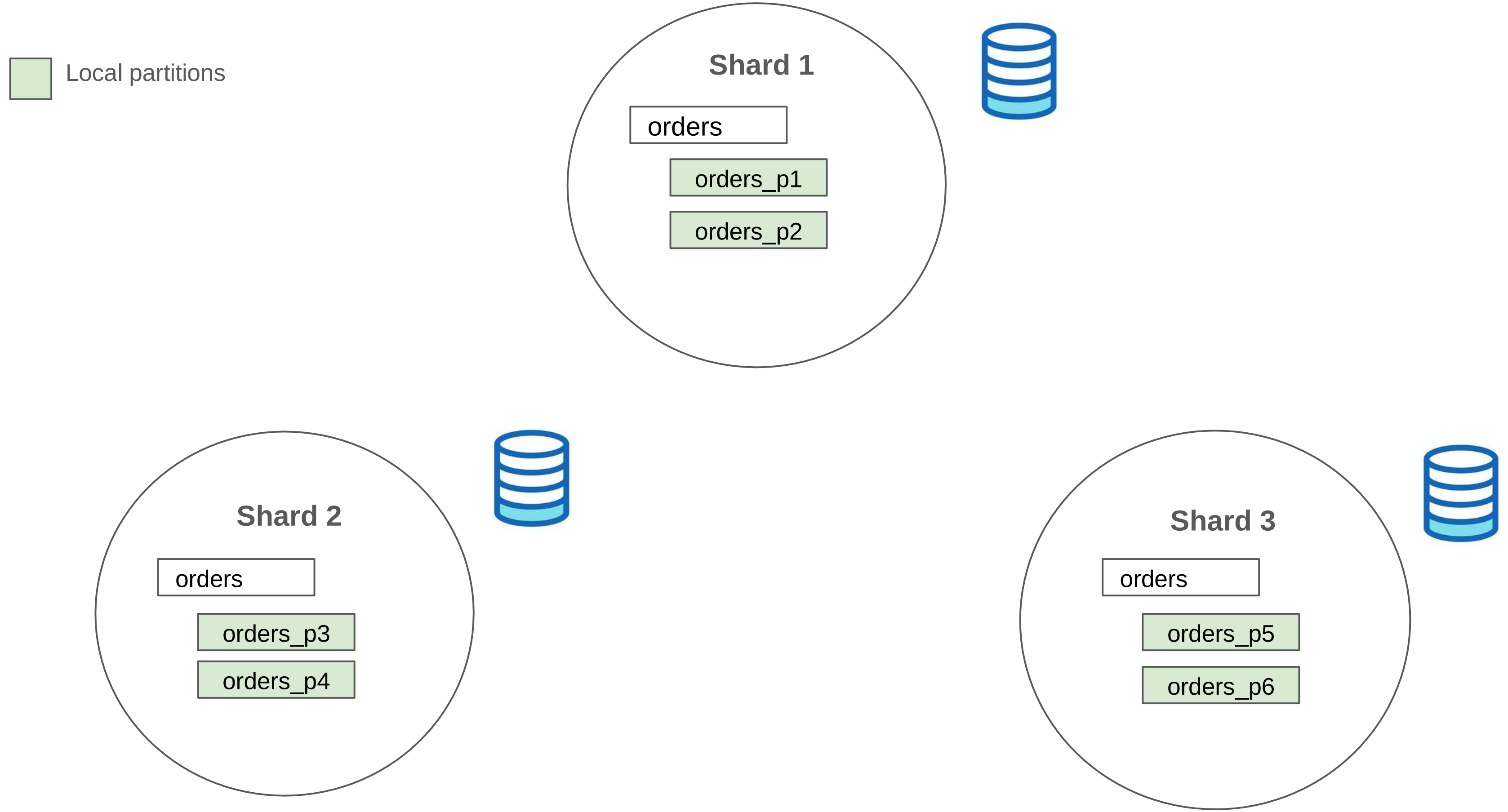
GRIDLEX

Sharding

- When you have lot of data in a single database and you are running out of Compute power or Storage, you can divide the data into separate database based on a **shard key** . This process is Known as **Sharding.**
- For example, here **customer_id** can be **shard key.**
- Types of sharding
    - **Hash based Sharding**
    - **Range based Sharding**
    - **Directory Sharding**
    - **Geo Sharding**

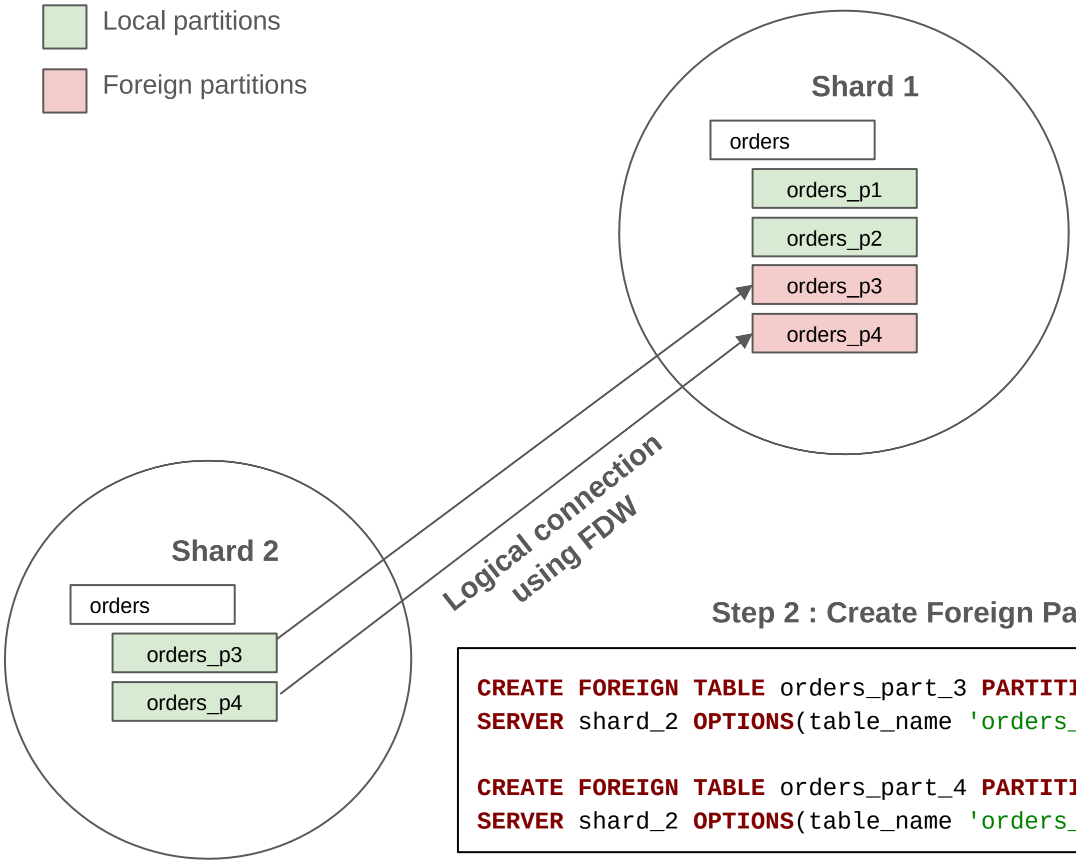
GRIDLEX

Sharding With Declarative Partitioning



GRIDLEX

Foreign Data Wrapper with Declarative Partitioning



Step 1 : Establishing connection to foreign server

```
create extension postgres_fdw;

create server shard_2
foreign data wrapper postgres_fdw
options (host 'localhost',dbname 'shard_2',port '5432');

create user mapping for shard1_user server shard_2 options
(user 'shard2_user' password 'postgres');
```

Step 2 : Create Foreign Partitions

```
CREATE FOREIGN TABLE orders_part_3 PARTITION OF orders FOR VALUES WITH (modulus 4 , remainder 2)
SERVER shard_2 OPTIONS(table_name 'orders_part_3');

CREATE FOREIGN TABLE orders_part_4 PARTITION OF orders FOR VALUES WITH (modulus 4 , remainder 3)
SERVER shard_2 OPTIONS(table_name 'orders_part_4');
```

GRIDLEX

Query plans

On Shard 1 :

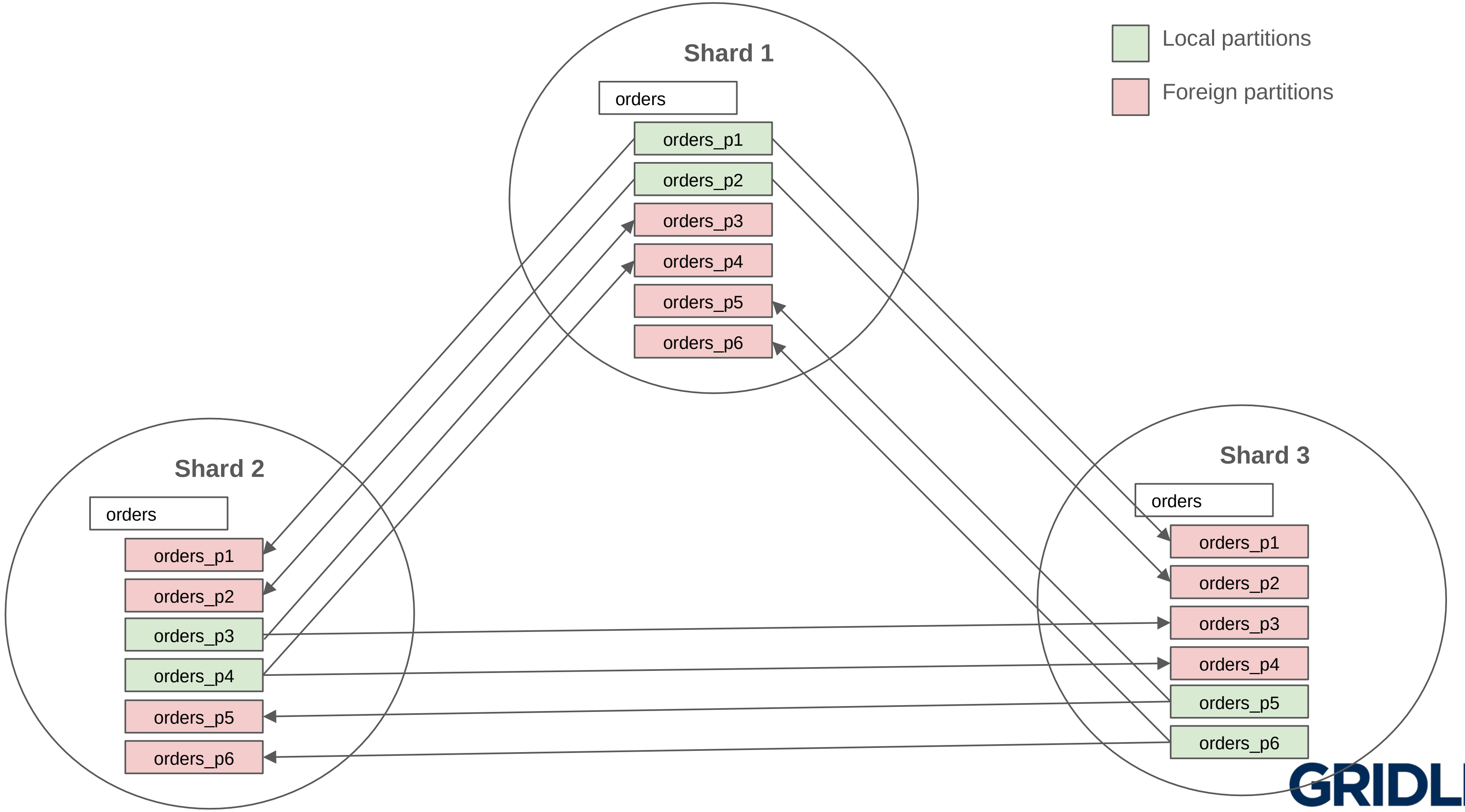
```sql
explain analyse
select * from orders;
```

QUERY PLAN
Append (cost=0.00..204.10 rows=4 width=19) (actual time=0.010..0.778 rows=4 loops=1)
-> Seq Scan on orders_part_1 orders_1 (cost=0.00..1.01 rows=1 width=18) (actual time=0.009..0.010 rows=1 loops=1)
-> Seq Scan on orders_part_2 orders_2 (cost=0.00..1.01 rows=1 width=19) (actual time=0.003..0.003 rows=1 loops=1)
-> Foreign Scan on orders_part_3 orders_3 (cost=100.00..101.03 rows=1 width=20) (actual time=0.512..0.513 rows=1 loops=1)
-> Foreign Scan on orders_part_4 orders_4 (cost=100.00..101.03 rows=1 width=19) (actual time=0.246..0.247 rows=1 loops=1)
Planning Time: 0.388 ms
Execution Time: 1.202 ms

```sql
explain analyse
select * from orders where customer_id = 212;
```

QUERY PLAN
Foreign Scan on orders_part_3 orders (cost=100.00..101.03 rows=1 width=20) (actual time=0.238..0.238 rows=1 loops=1)
Planning Time: 0.089 ms
Execution Time: 0.427 ms

Foreign Data Wrapper with Declarative Partitioning



Postgres FDW Performance optimization capabilities

- Supports SELECT , INSERT, UPDATE, DELETE, COPY or TRUNCATE.
- **Cross version compatibility** - it supports upto version 8.3.
- **Capability to pushdown** - Filters, Joins, aggregate functions, case expressions to foreign tables.
- **Partition wise join** based on the partition boundaries.
- **Async Parallel execution** capability
- Limited push down of ORDER BY and LIMIT.

Postgres FDW tuning parameters

- Remote execution
    - **Fetch_size** : A cursor will be declared when a remote transaction is started. We can change this parameter to find the best fit. Default is 100.
- Cost estimation
    - **use_remote_estimate** :
        - This parameter can be set to TRUE to get the statistics from the remote database before generating the query plan, but it adds up to the cost.
    - **Fdw_startup_cost** : cost associated with the connection establishment to remote server. Default is 100.
    - **Fdw_tuple_cost** : network cost associated with fetching a single row from remote database. Default is 0.1

Async / Parallel Execution Capability

1. **Set async_capable to TRUE** for foreign servers, which enables queries to run foreign scans asynchronously.
2. We can do this at foreign server level or table level. Table level will replace server level properties.

```
postgres=# alter server db1 options (add async_capable 'true');
ALTER SERVER

postgres=# explain (analyze, buffers, verbose)
select order_id, order_date, qty*price total_price from orders ;
                                                                 QUERY PLAN
----------------------------------------------------------------------------------------------------
 Append  (cost=0.00..180.08 rows=1927 width=44) (actual time=2.409..4.899 rows=18 loops=1)
   Buffers: shared hit=1
   ->  Seq Scan on public.orders_p1 orders_1  (cost=0.00..22.75 rows=850 width=44)
                                              (actual time=0.029..0.143 rows=9 loops=1)
         Output: orders_1.order_id, orders_1.order_date, ((orders_1.qty)::numeric * orders_1.price)
         Buffers: shared hit=1
   ->  Async Foreign Scan on public.orders_p2 orders_2  (cost=100.00..147.69 rows=1077 width=44)
                                                        (actual time=2.506..2.598 rows=9 loops=1)
         Output: orders_2.order_id, orders_2.order_date, ((orders_2.qty)::numeric * orders_2.price)
         Remote SQL: SELECT order_id, order_date, qty, price FROM public.orders_p2
 Query Identifier: -4744352895563044555
 Planning Time: 0.154 ms
 Execution Time: 9.476 ms
(11 rows)
```

Joins and predicates pushdown capability

1. **set enable_partitionwise_join = on;**
2. Here in this case both foreign tables are partitioned based on same boundaries , so join is pushed to remote server and partionwise join is performed and then, they are appended.
3. Predicate (filters) are also pushed down to foreign server.

```
postgres=# set enable_partitionwise_join = on;
SET
postgres=# explain (verbose, costs off) SELECT a1.* FROM parent_local a1 join fk_local b on a1.a = b.a where a1.c = 'mytest';
                                                                    QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------
----------
 Append
   ->  Async Foreign Scan
         Output: a1_1.a, a1_1.b, a1_1.c, a1_1.d
         Relations: (public.parent_remote1 a1_1) INNER JOIN (public.fk_remote1 b_1)
         Remote SQL: SELECT r4.a, r4.b, r4.c, r4.d FROM (public.child_local1 r4 INNER JOIN public.fk_local1 r6 ON (((r4.a = r6.a)) AND ((r4.c =
'mytest'::text))))
   ->  Async Foreign Scan
         Output: a1_2.a, a1_2.b, a1_2.c, a1_2.d
         Relations: (public.parent_remote2 a1_2) INNER JOIN (public.fk_remote2 b_2)
         Remote SQL: SELECT r5.a, r5.b, r5.c, r5.d FROM (public.child_local2 r5 INNER JOIN public.fk_local2 r7 ON (((r5.a = r7.a)) AND ((r5.c =
'mytest'::text))))
(9 rows)
```

GRIDLEX

Aggregate function push down Capability

1. **set enable_partitionwise_aggregate = on;**
2. It enables partition wise partial aggregate and then do the total aggregate. Partition wise aggregate is also pushed down to foreign server.

```
postgres=# set enable_partitionwise_aggregate = 'on' ;
SET
postgres=# explain (analyze, buffers, verbose)
select asset_id, min(min_value), max(max_value), sum(volume_value) from asset group by asset_id ;
                                                     QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------
 Append  (cost=106.40..155.40 rows=400 width=104) (actual time=3.460..6.055 rows=2 loops=1)
   Buffers: shared hit=1
   ->  Async Foreign Scan  (cost=106.40..129.30 rows=200 width=104) (actual time=3.398..3.415 rows=1 loops=1)
         Output: asset.asset_id, (min(asset.min_value)), (max(asset.max_value)), (sum(asset.volume_value))
         Relations: Aggregate on (public.asset1to100000 asset)
         Remote SQL: SELECT asset_id, min(min_value), max(max_value), sum(volume_value) FROM public.asset1to100000 GROUP BY 1
   ->  HashAggregate  (cost=21.60..24.10 rows=200 width=104) (actual time=0.116..0.153 rows=1 loops=1)
         Output: asset_1.asset_id, min(asset_1.min_value), max(asset_1.max_value), sum(asset_1.volume_value)
         Group Key: asset_1.asset_id
         Batches: 1  Memory Usage: 40kB
         Buffers: shared hit=1
         ->  Seq Scan on public.asset100001to200000 asset_1  (cost=0.00..15.80 rows=580 width=104)
                                                           (actual time=0.045..0.074 rows=2 loops=1)
               Output: asset_1.asset_id, asset_1.min_value, asset_1.max_value, asset_1.volume_value
               Buffers: shared hit=1
 Query Identifier: 8742586258246434012
 Planning Time: 0.351 ms
 Execution Time: 11.722 ms
(17 rows)
```

GRIDLEX

Limitations on ORDER BY and LIMIT

1. When using partitioning with fdw , for a query containing order_by followed by limit on foreign partition; order_by is pushed down to the foreign server and all the rows are returned and then the limit is applied in local server.
2. But when using fdw without partitioning on a foreign table both order by and limit is pushed to the foreign server.

```
postgres=# explain (analyze, buffers, verbose) select asset_id, mkt_close_date, max_value, volume_value from asset order by max_value desc limit 1;
                                                                 QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------------------------------
 Limit  (cost=130.31..130.35 rows=1 width=80) (actual time=18.066..18.149 rows=1 loops=1)
   Output: asset.asset_id, asset.mkt_close_date, asset.max_value, asset.volume_value
   Buffers: shared hit=1
   ->  Merge Append  (cost=130.31..182.61 rows=1367 width=80) (actual time=18.037..18.095 rows=1 loops=1)
         Sort Key: asset.max_value DESC
         Buffers: shared hit=1
         ->  Foreign Scan on public.asset1to100000 asset_1  (cost=100.00..137.18 rows=787 width=80) (actual time=17.269..17.278 rows=1 loops=1)
               Output: asset_1.asset_id, asset_1.mkt_close_date, asset_1.max_value, asset_1.volume_value
               Remote SQL: SELECT asset_id, mkt_close_date, max_value, volume_value FROM public.asset1to100000 ORDER BY max_value DESC NULLS FIRST
         ->  Sort  (cost=18.70..20.15 rows=580 width=80) (actual time=0.714..0.738 rows=1 loops=1)
               Output: asset_2.asset_id, asset_2.mkt_close_date, asset_2.max_value, asset_2.volume_value
               Sort Key: asset_2.max_value DESC
               Sort Method: top-N heapsort  Memory: 25kB
               Buffers: shared hit=1
               ->  Seq Scan on public.asset100001to200000 asset_2  (cost=0.00..15.80 rows=580 width=80) (actual time=0.018..0.304 rows=33 loops=1)
                     Output: asset_2.asset_id, asset_2.mkt_close_date, asset_2.max_value, asset_2.volume_value
                     Buffers: shared hit=1
 Query Identifier: 6898531713158321828
 Planning Time: 0.178 ms
 Execution Time: 23.602 ms
(20 rows)
```

```
postgres=# explain (analyze, buffers, verbose) select asset_id, mkt_close_date, max_value, volume_value
from public.asset100001to200000 order by max_value desc limit 1;
                                  QUERY PLAN
-------------------------------------------------------------------------------------------
 Foreign Scan on public.asset100001to200000  (cost=100.00..100.07 rows=1 width=38)
                                              (actual time=8.252..8.270 rows=1 loops=1)
   Output: asset_id, mkt_close_date, max_value, volume_value
   Remote SQL: SELECT asset_id, mkt_close_date, max_value, volume_value
               FROM public.asset100001to200000 ORDER BY max_value DESC NULLS FIRST LIMIT 1::bigint
 Query Identifier: 5301853017547915870
 Planning Time: 0.138 ms
 Execution Time: 13.122 ms
(6 rows)
```

What are different isolation levels in Postgres ?

GRIDLEX

What are different isolation levels in Postgres ?

1. Read Uncommitted
2. **Read Committed**
3. **Repeatable Reads**
4. Serializable

Read Committed vs Repeatable Reads

Transaction A	Transaction B
read order (where id =1) (1, book, $50, ‘John’)	
	update order set amount = $100 (where id =1)
	commit
read order (where id =1) (1, book, $100, ‘John’)	

Transaction A will get same record different values

Read Committed

Transaction A	Transaction B
read order (where id =1) (1, book, $50, ‘John’)	
	update order set amount = $100 (where id =1)
	commit
read order (where id =1) (1, book, $50, ‘John’)	

Transaction A will get the record with same values between different reads

Repeatable Reads

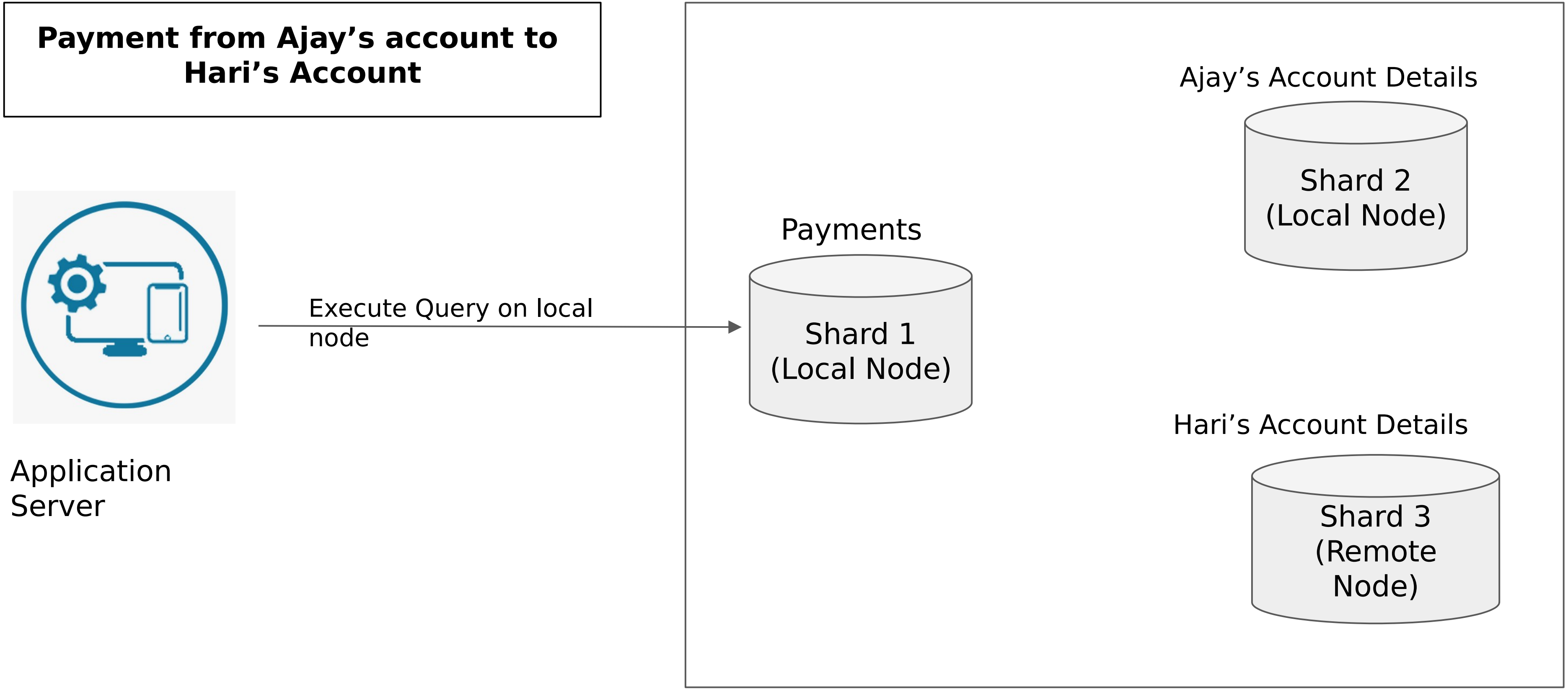
GRIDLEX

Isolation Levels for Foreign Data Wrapper

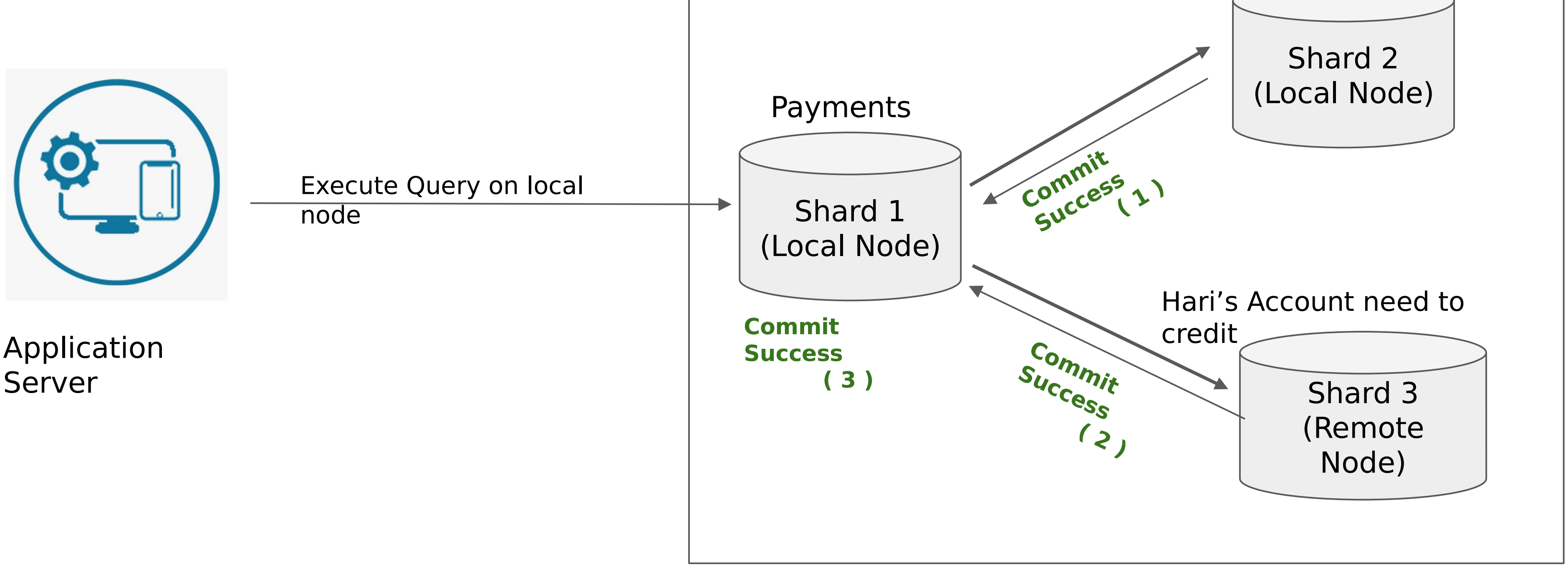
- In general postgres transactions have default **Read committed** isolation level in postgreSQL.
- For foreign nodes it will always use **Repeatable reads** isolation level , so a snapshot is taken at the beginning of transaction and it uses that throughout the transaction.
- This choice ensures that if a query performs multiple table scans on the remote server, it will get snapshot-consistent results for all the scans. A consequence is that successive queries within a single transaction will see the same data from the remote server, even if concurrent updates are occurring on the remote server due to other activities.

Atomicity using Foreign Data Wrappers

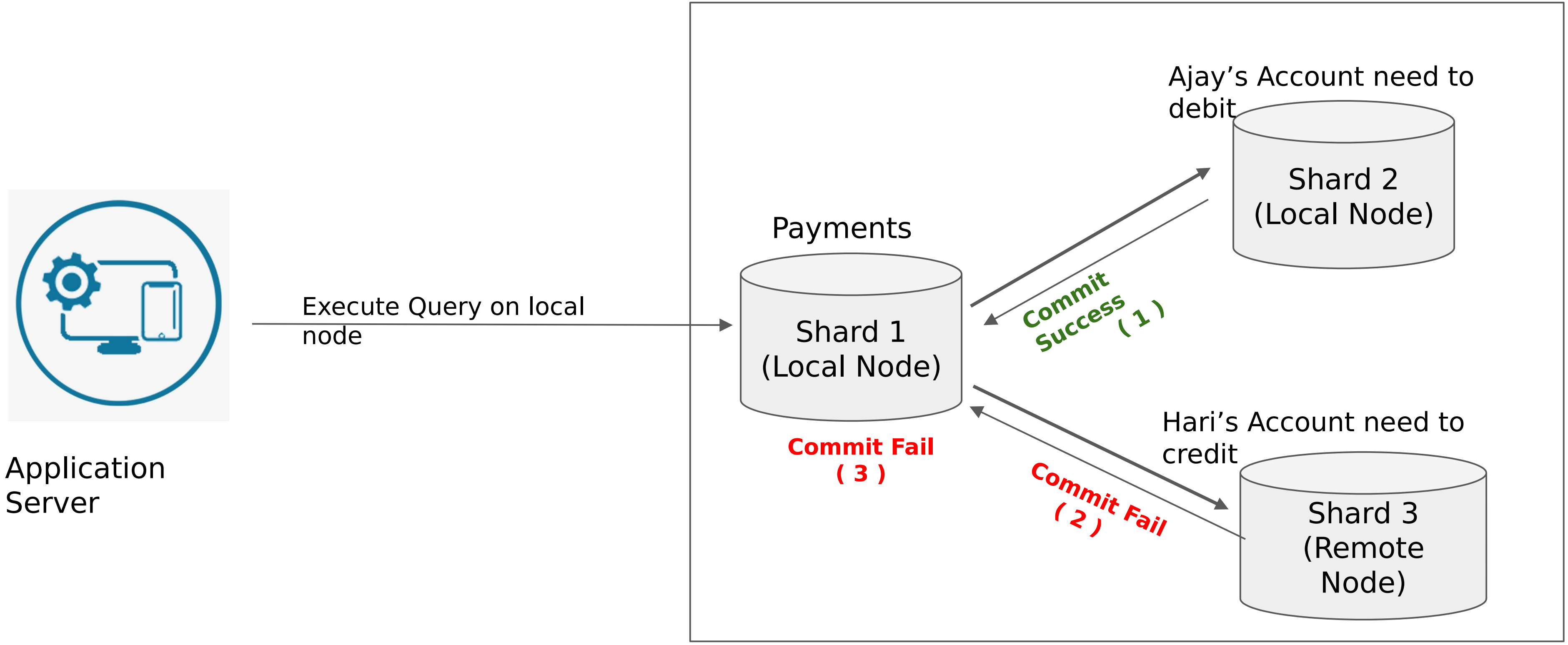
Atomicity in Distributed Transactions



Payment from Ajay’s account to Hari’s Account



Atomicity in Distributed Transactions

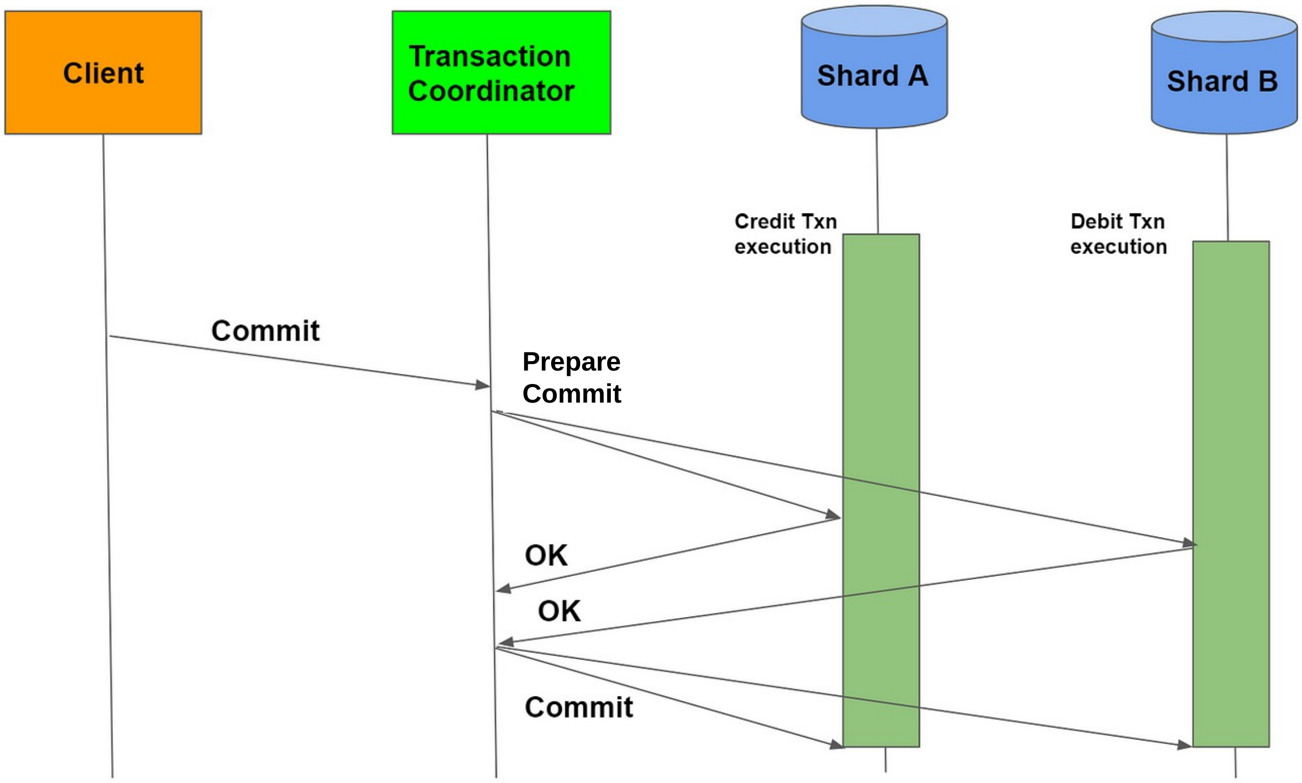


In this case data will be in inconsistent state because Ajay's Account is debited but hari's account is not credited

GRIDLEX

How Atomicity can be achieved in Distributed Transactions

2 phase Commit



GRIDLEX

2 phase Commit

1. Prepare Phase

The prepare phase begins when a coordinator initiates a transaction. After making the necessary local changes, the coordinator sends a prepare message to all participant nodes, instructing them to prepare to commit the transaction.

Each participant executes the transaction locally, writes the changes to a log, and responds to the coordinator. If the transaction executes successfully, the participant votes to commit and enters a prepared state. If the transaction fails at any participant node, it votes to abort.

2. Commit Phase

In the commit phase, the coordinator collects votes from all participant nodes. If all participants vote to commit, the coordinator writes a commit record in its log and sends a global commit message to all participants. Each participant then commits the transaction locally and sends an acknowledgment to the coordinator.

However, if any participant votes to abort or if the coordinator doesn't receive a response from a participant (a timeout occurs), it decides to abort the transaction. The coordinator logs the abort record and sends a global abort message to all participants. Each participant then undoes the transaction changes and sends an acknowledgment to the coordinator.

Limitations

- **Transaction atomicity** :
    - No global transaction mechanism like 2 phase commit
- **Deadlock detection**
    - For distributed transactions, when two or more shards are involved in a deadlock then it won't be recognised.
- **Index Pushdown for Functional and Casted columns**
    - For example, I have field called customer_name which is case sensitive but when sorting on customer_name I want that field to be case insensitive.
    - B-tree index on (OrgID, lower(customer_name)).
    - In this case Index pushdown doesn't work for sort queries

FDW is being improved from version to version, these problems may be addressed in next releases.

Alternatives

- Application level sharding
  - https://aws.amazon.com/blogs/database/sharding-with-amazon-relational-database-service/
- Extensions like Citus for automatic sharding
  - Citus is fully open source with features like automatic shard rebalancing.
  - Citus has Global Transaction Management and Deadlock Detection.
  - Citus is not supported in AWS RDS and Aurora.

GRIDLEX

How do you handle database sharding in your application ?

Any Queries

Thanks