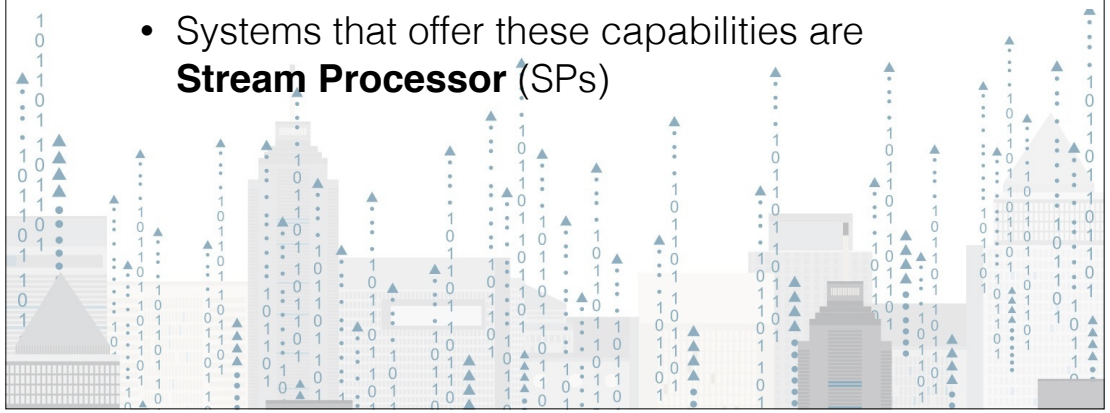
FlowDB

Integrating Stream Processing and Consistent State Management

Lorenzo Affetti
Alessandro Margara
Gianpaolo Cugola

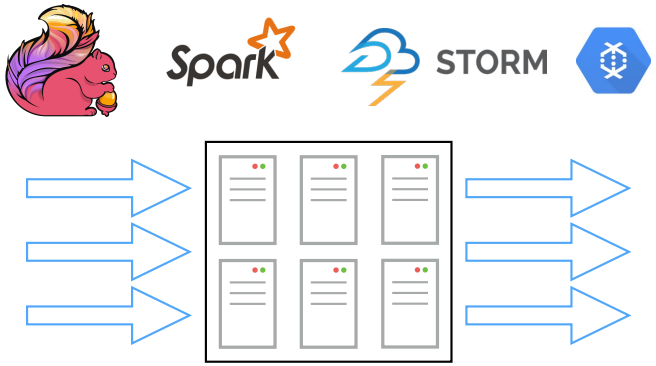
The Problem

- We are in the **fast data** era
- Companies need **continuous processing** of data and **continuous production of results**
- Near **real-time processing**
- Systems that offer these capabilities are **Stream Processor** (SPs)



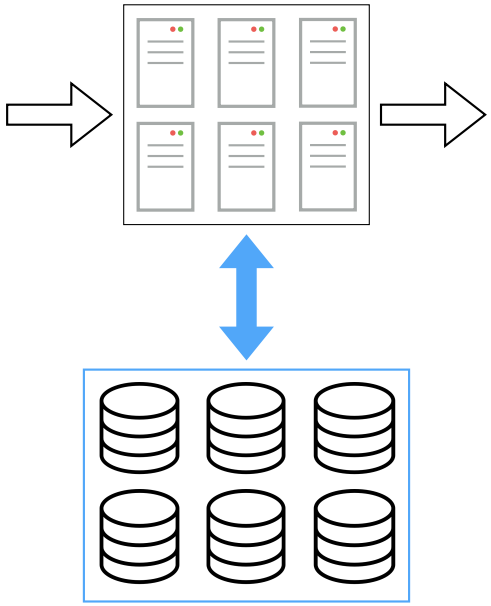
The Problem

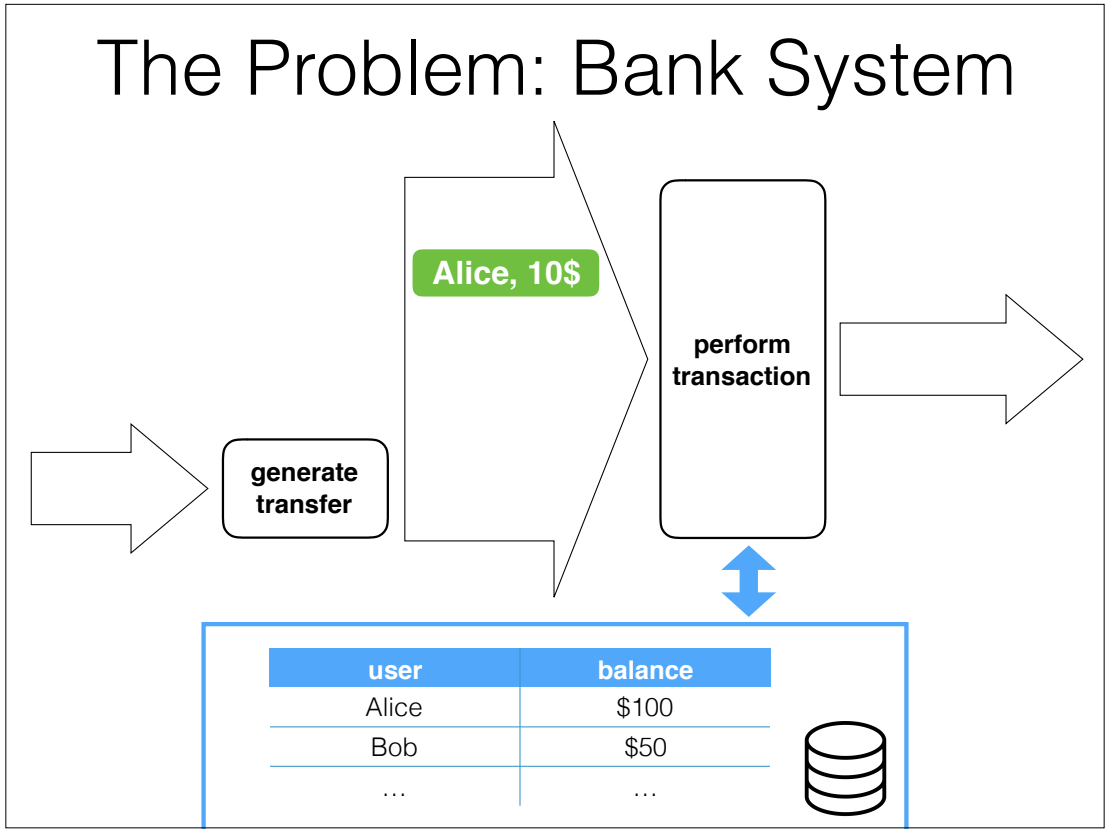
Modern SPs are **distributed** in order to cope with the volume and velocity of data



The Problem: Global State

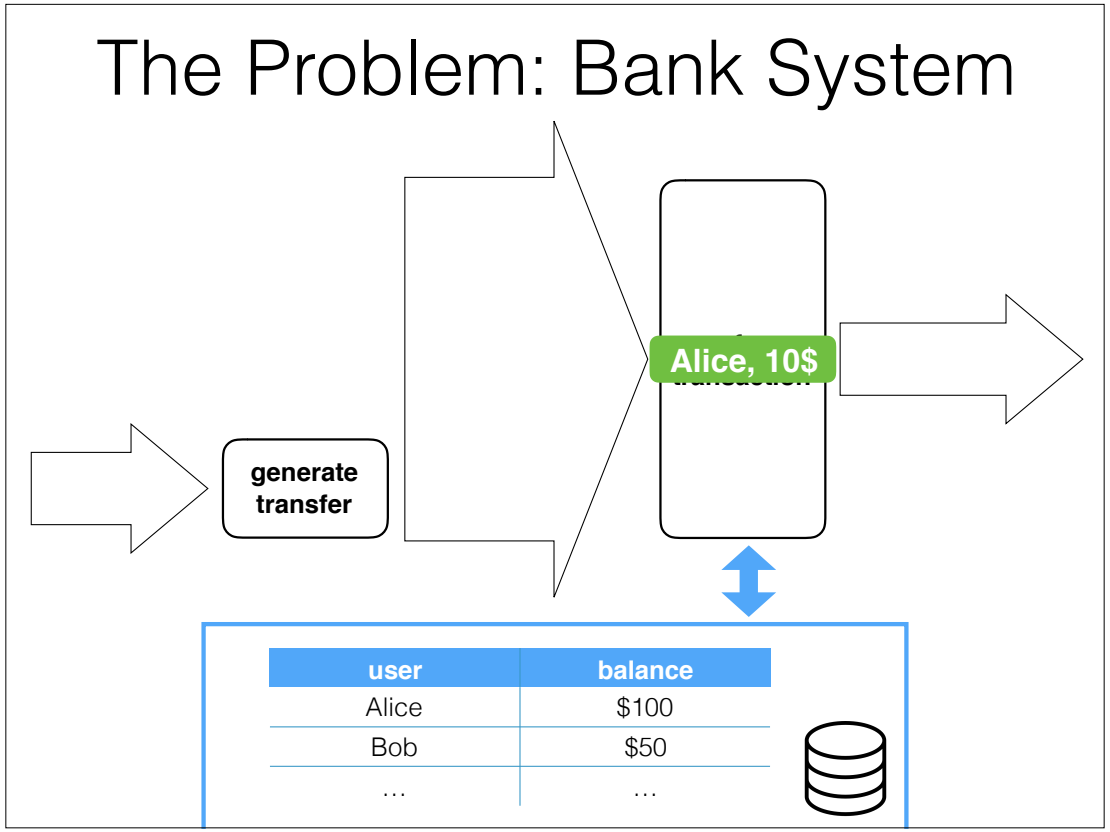
- Stream processing applications process data and **change some global state**
- The state is usually stored in other systems
- This fact introduces **resource overhead**
- And complicates the deploy of the application





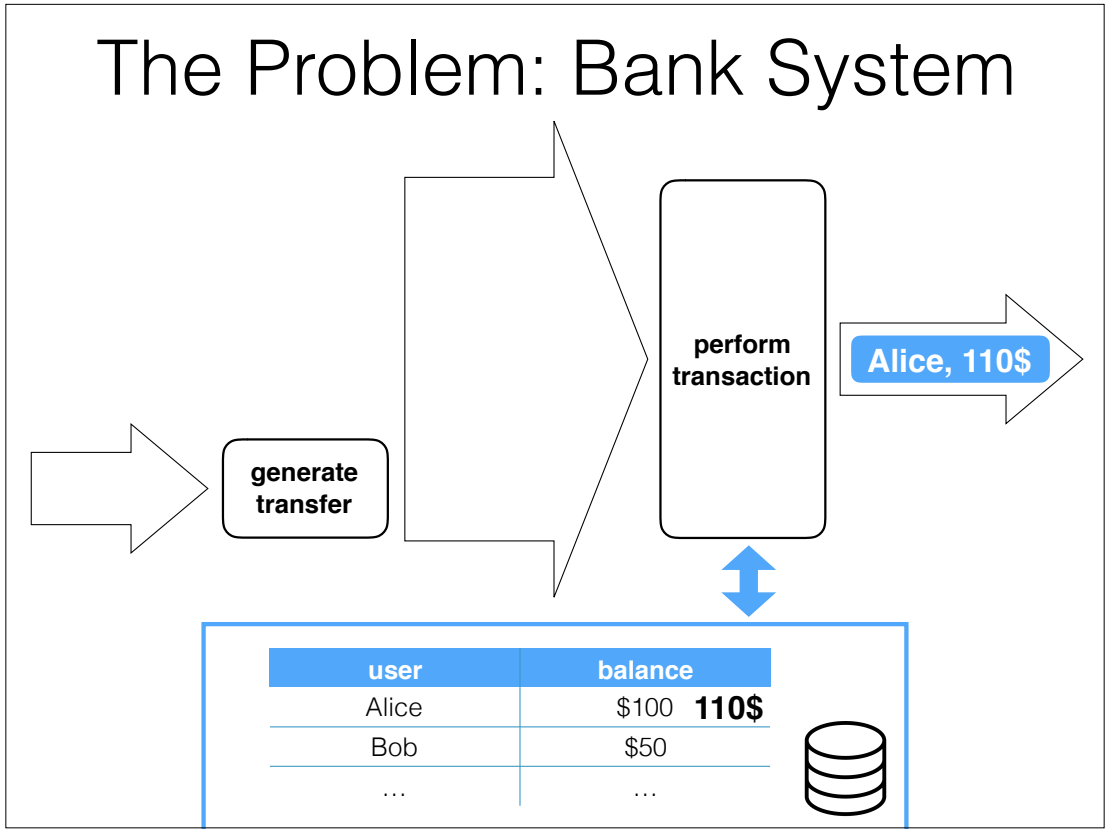
Every new request that enters the SP application performs a request to the database.

However, the database is not built to cope with the requests rate that a SP il built for and it becomes the bottleneck.



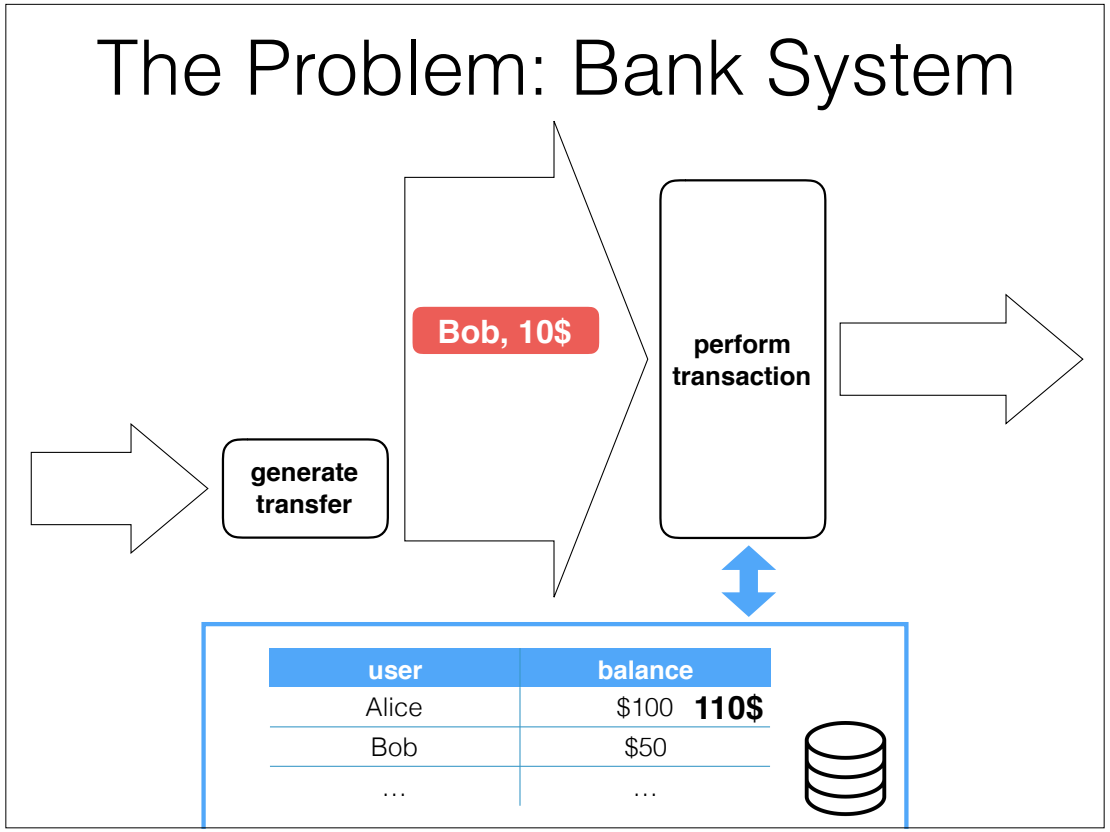
Every new request that enters the SP application performs a request to the database.

However, the database is not built to cope with the requests rate that a SP il built for and it becomes the bottleneck.



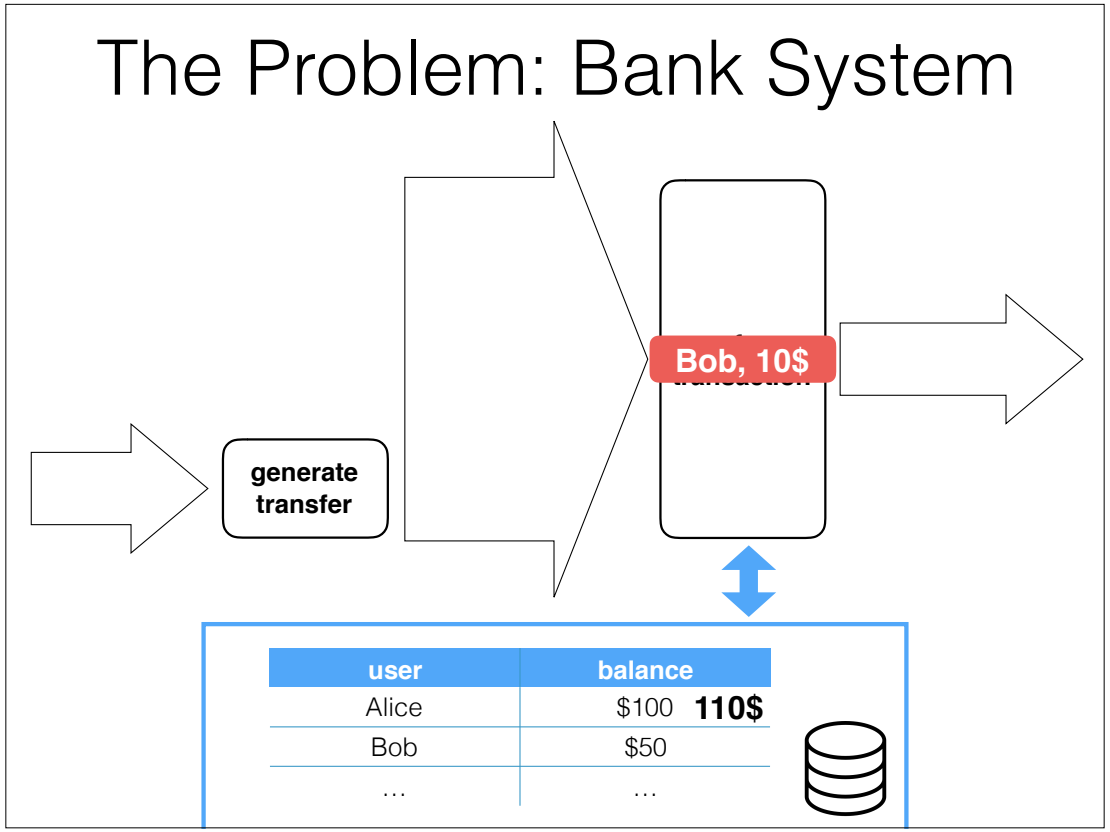
Every new request that enters the SP application performs a request to the database.

However, the database is not built to cope with the requests rate that a SP il built for and it becomes the bottleneck.



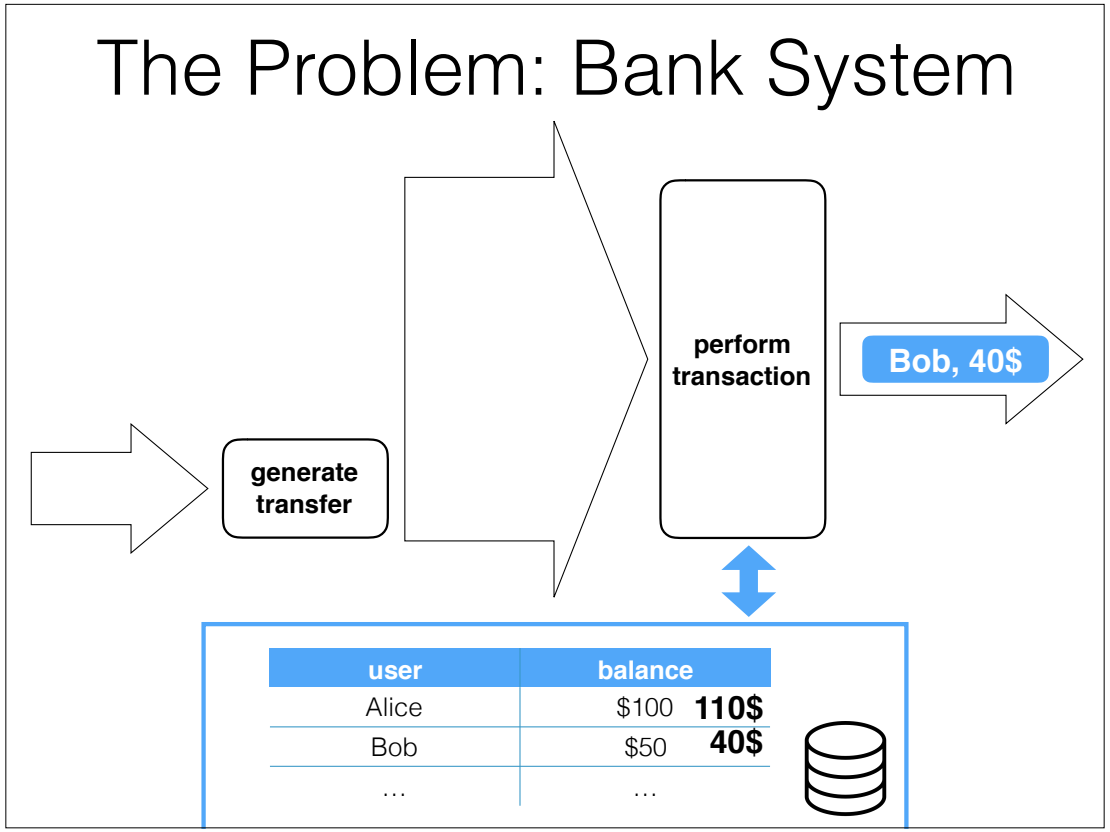
Every new request that enters the SP application performs a request to the database.

However, the database is not built to cope with the requests rate that a SP il built for and it becomes the bottleneck.



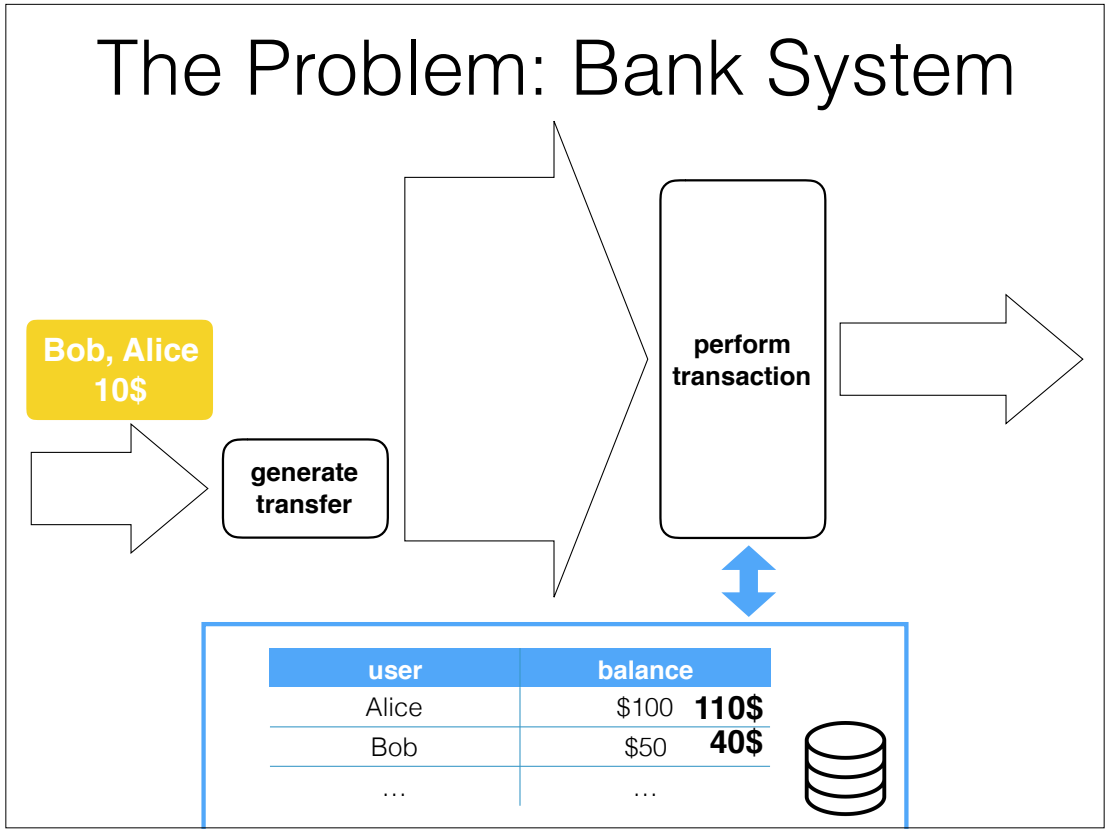
Every new request that enters the SP application performs a request to the database.

However, the database is not built to cope with the requests rate that a SP il built for and it becomes the bottleneck.



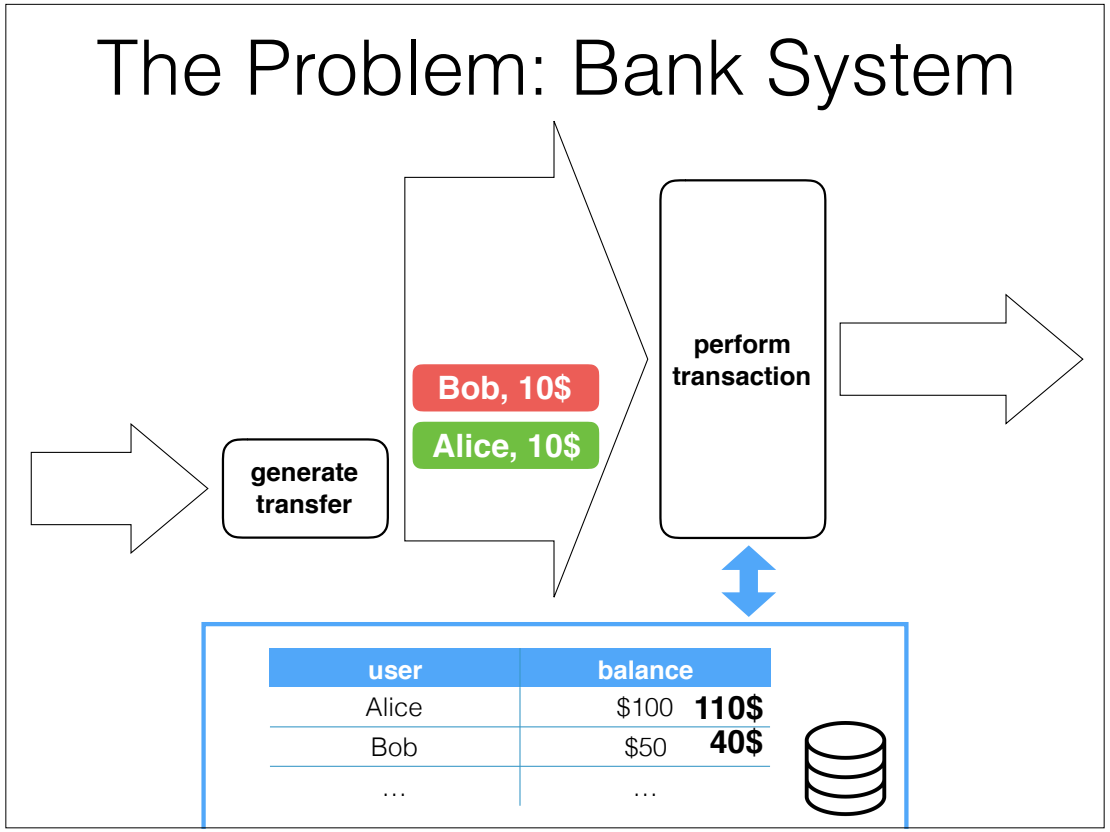
Every new request that enters the SP application performs a request to the database.

However, the database is not built to cope with the requests rate that a SP il built for and it becomes the bottleneck.



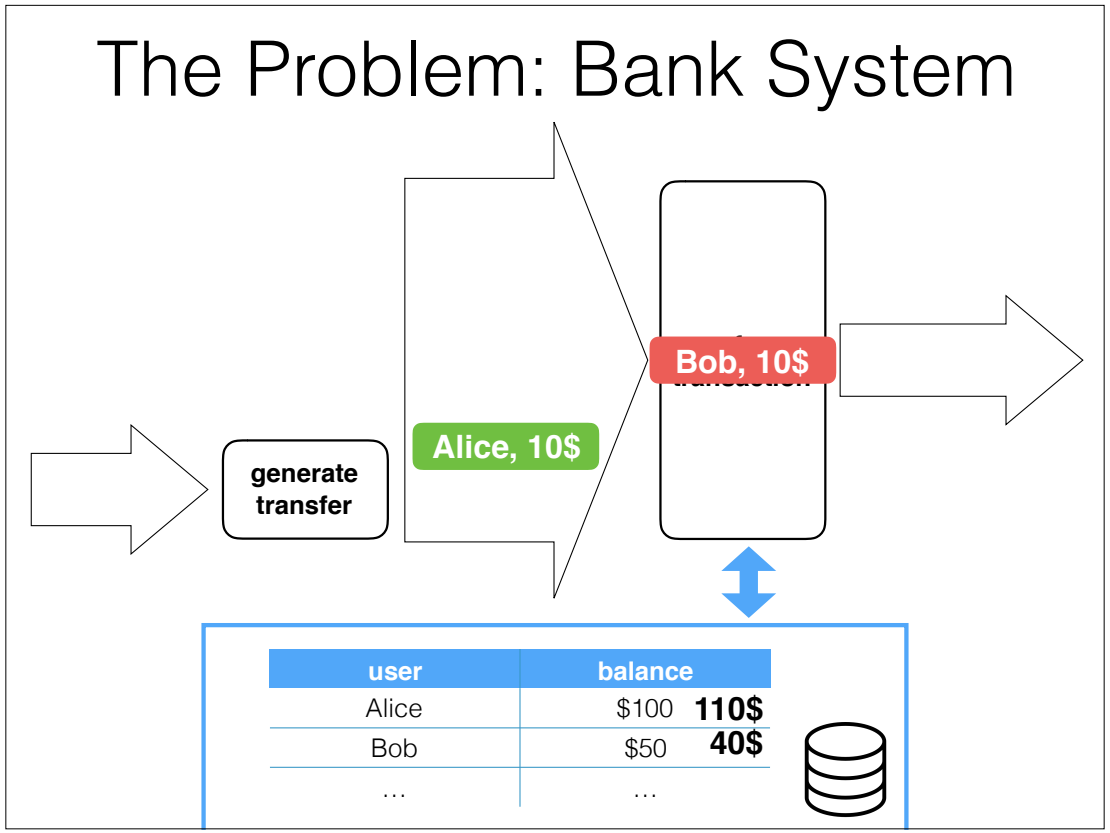
Every new request that enters the SP application performs a request to the database.

However, the database is not built to cope with the requests rate that a SP il built for and it becomes the bottleneck.



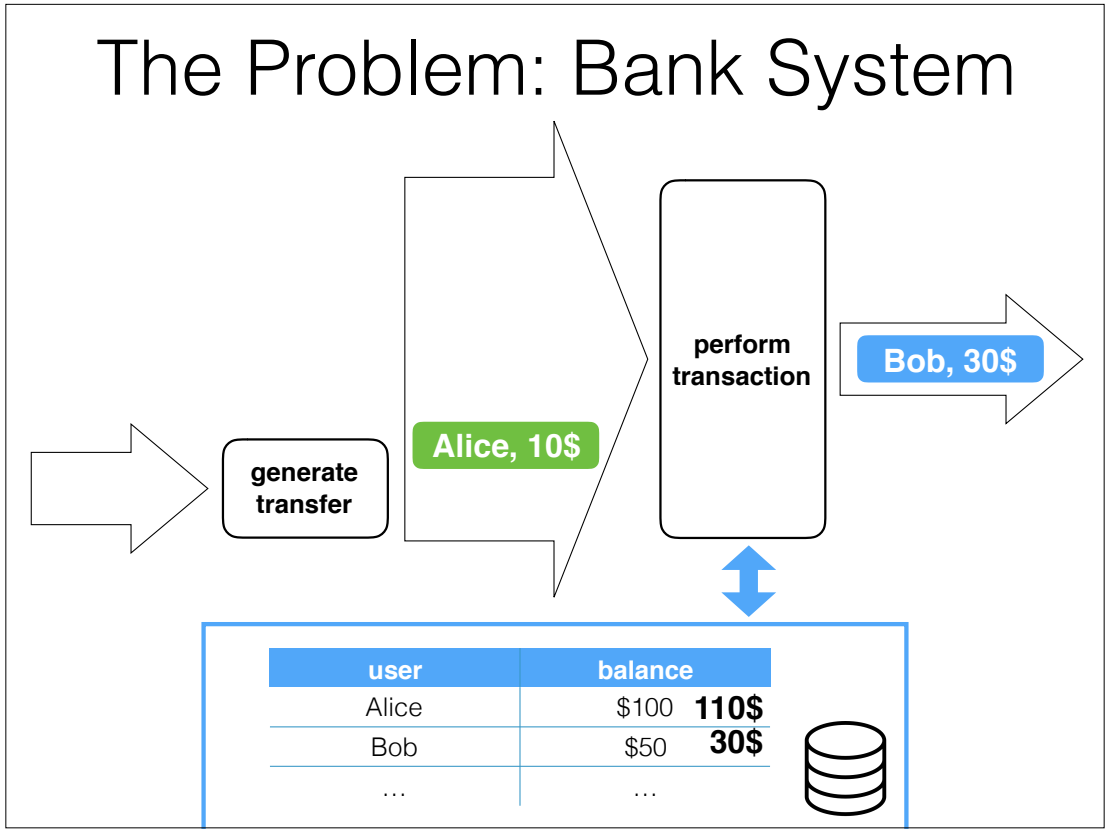
Every new request that enters the SP application performs a request to the database.

However, the database is not built to cope with the requests rate that a SP il built for and it becomes the bottleneck.



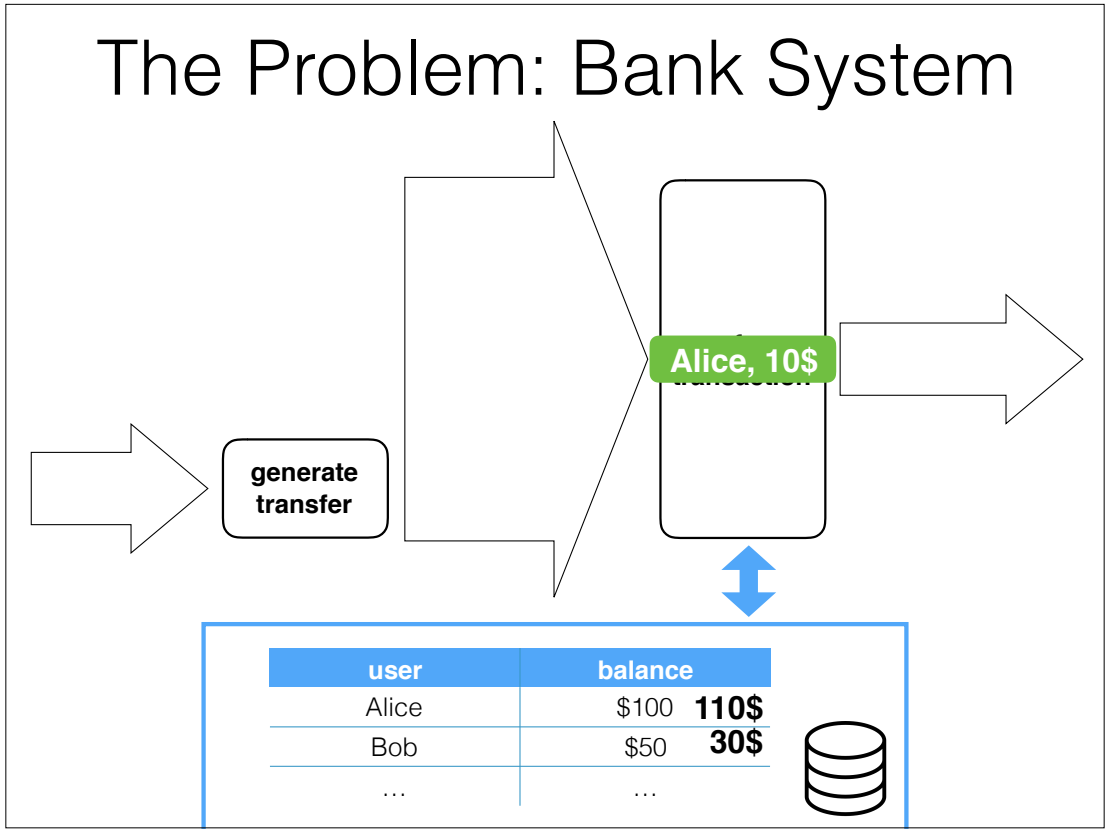
Every new request that enters the SP application performs a request to the database.

However, the database is not built to cope with the requests rate that a SP il built for and it becomes the bottleneck.



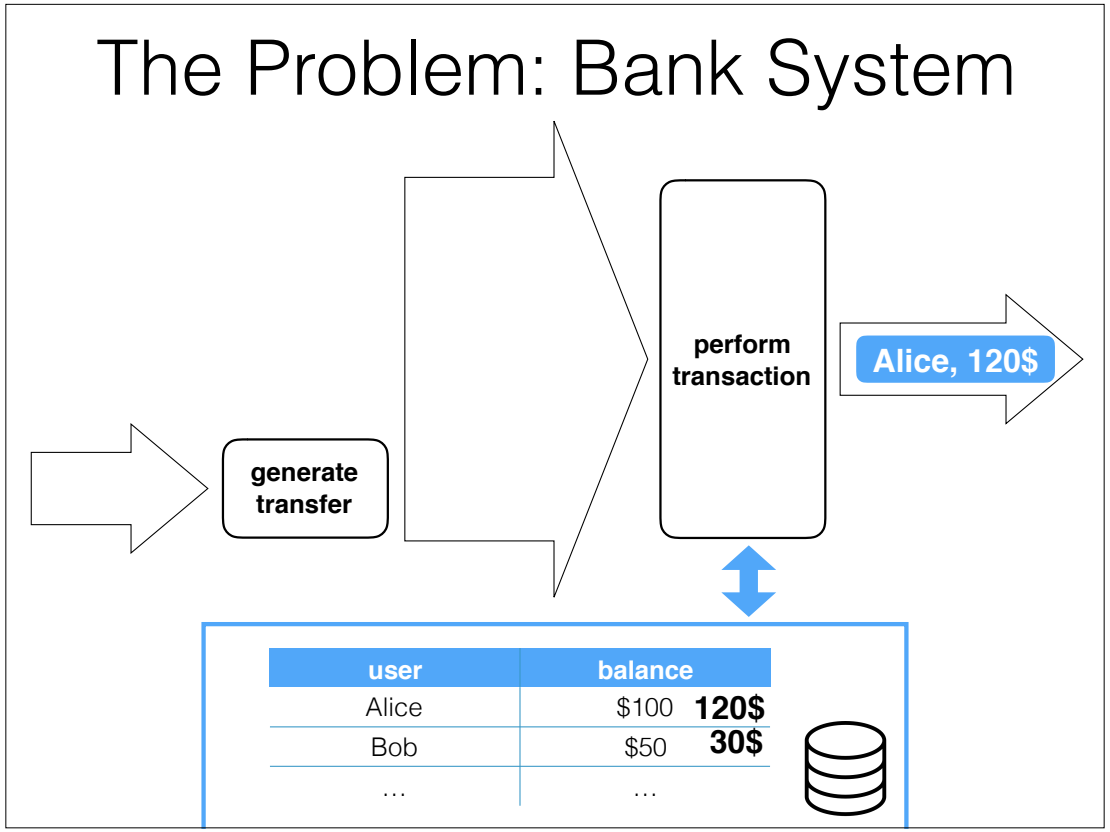
Every new request that enters the SP application performs a request to the database.

However, the database is not built to cope with the requests rate that a SP il built for and it becomes the bottleneck.



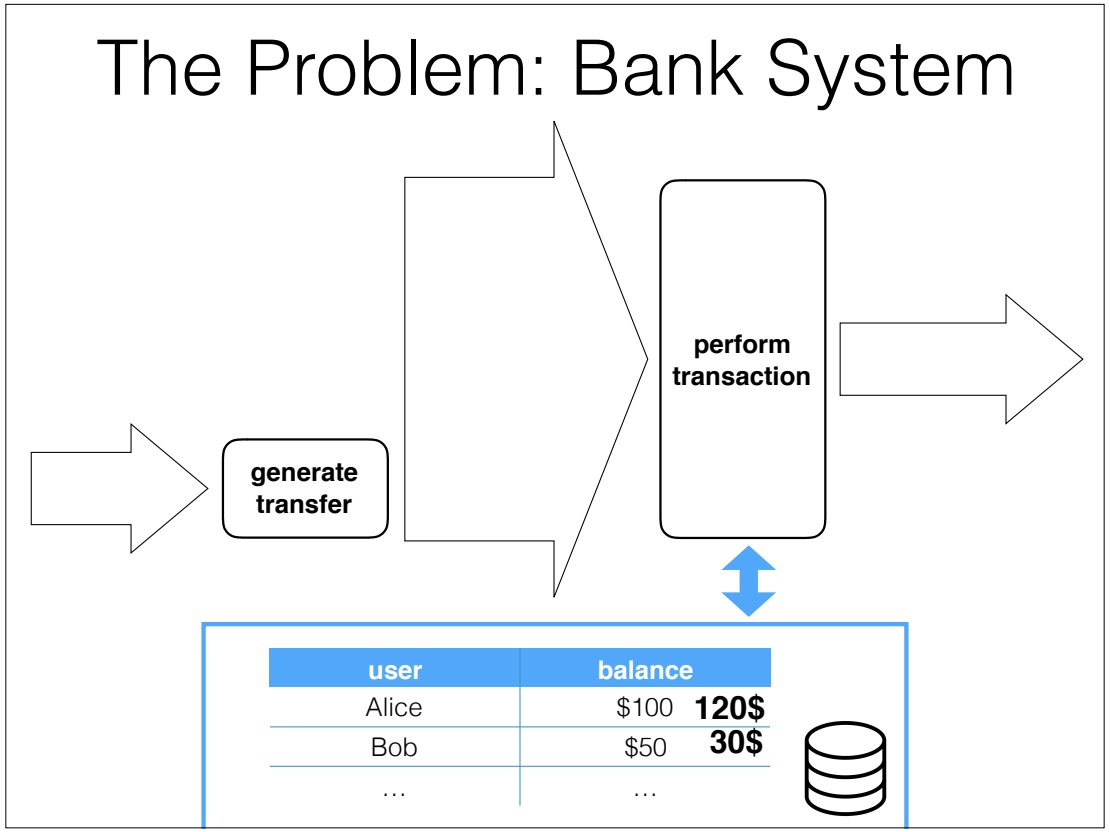
Every new request that enters the SP application performs a request to the database.

However, the database is not built to cope with the requests rate that a SP il built for and it becomes the bottleneck.



Every new request that enters the SP application performs a request to the database.

However, the database is not built to cope with the requests rate that a SP il built for and it becomes the bottleneck.

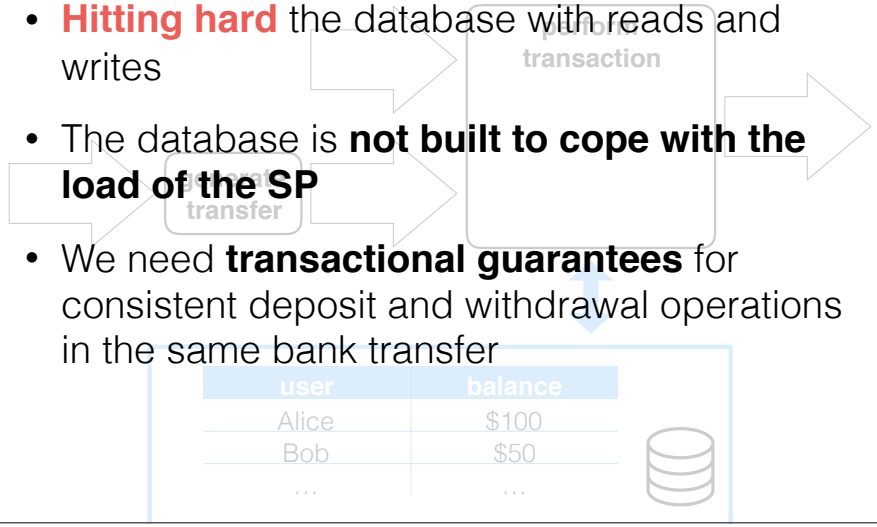


Every new request that enters the SP application performs a request to the database.

However, the database is not built to cope with the requests rate that a SP il built for and it becomes the bottleneck.

The Bank System: Issues

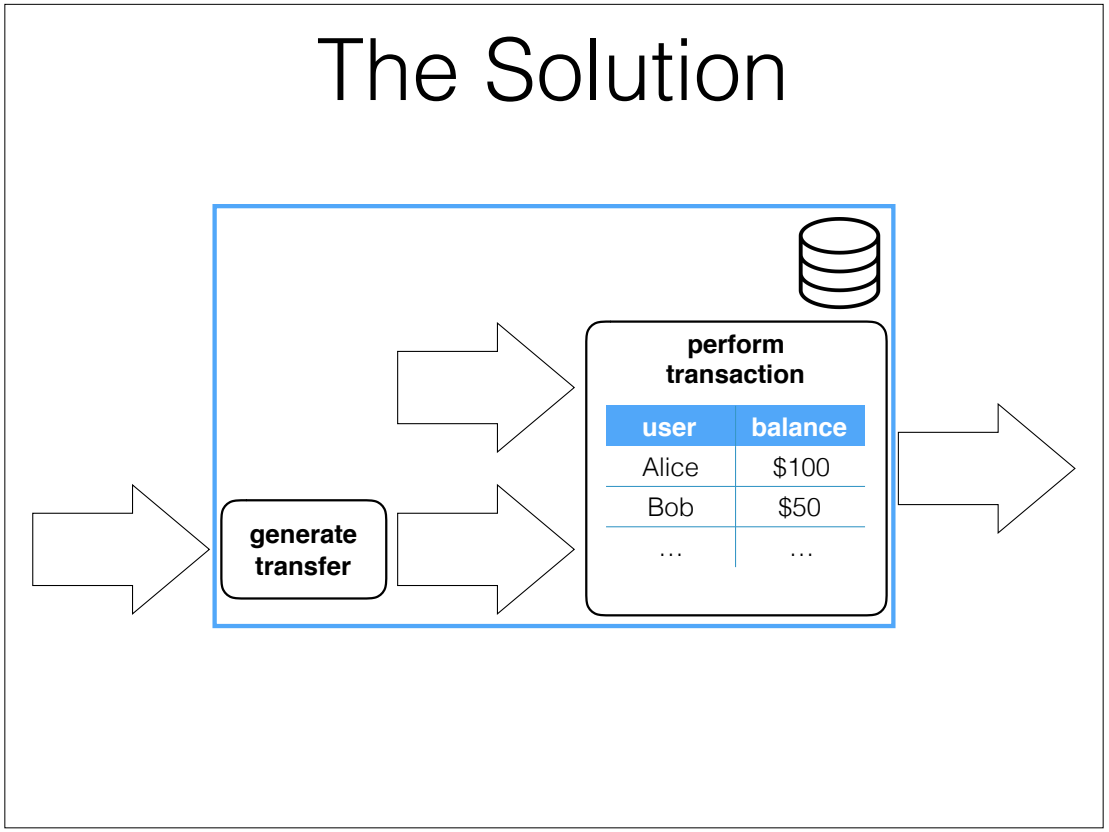
- **Hitting hard** the database with reads and writes
- The database is **not built to cope with the load of the SP**
- We need **transactional guarantees** for consistent deposit and withdrawal operations in the same bank transfer



The Problem: SP vs DB

- Built for **distributed computation**
- It keeps an **internal state**, but:
  - Not queryable
  - Not transactional

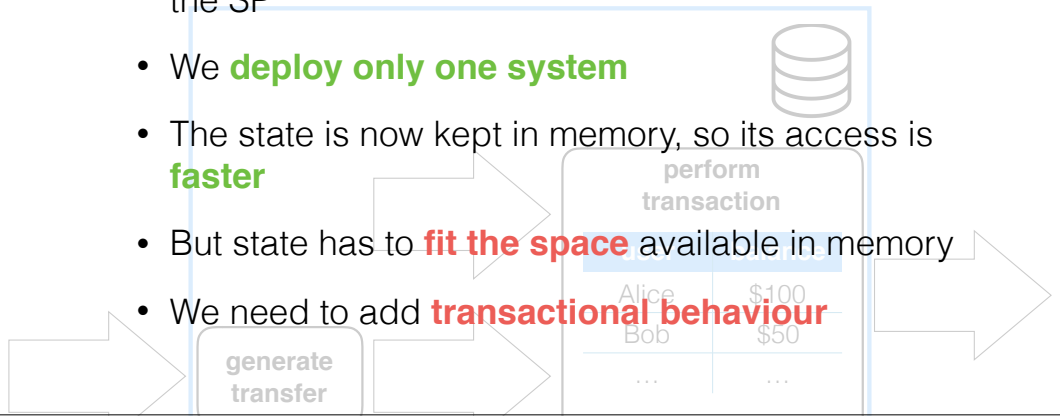
- Built for **transactional** behaviour
- It lacks of distributed computation capabilities



Our solution consists in embedding the database in the SP.

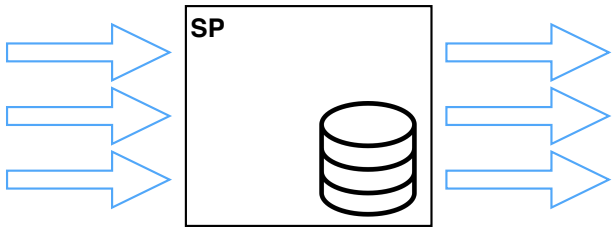
The Solution

- We **eliminated the bottleneck** for external communication
- We can fully leverage the **parallel execution** of the SP
- We **deploy only one system**
- The state is now kept in memory, so its access is **faster**
- But state has to **fit the space** available in memory
- We need to add **transactional behaviour**

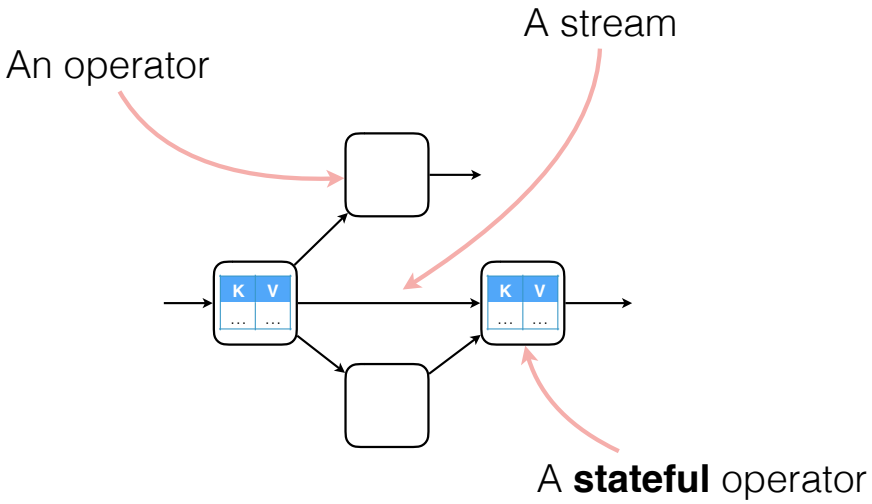


The Solution

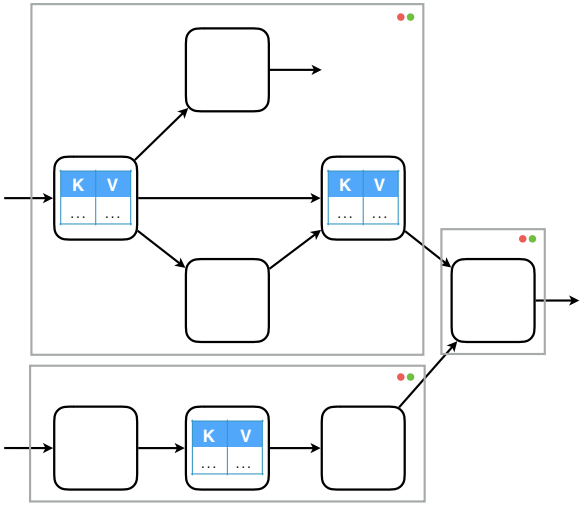
- Built for **distributed computation**
- Exposes **queryable state**
- Offers **transactional guarantees** on state



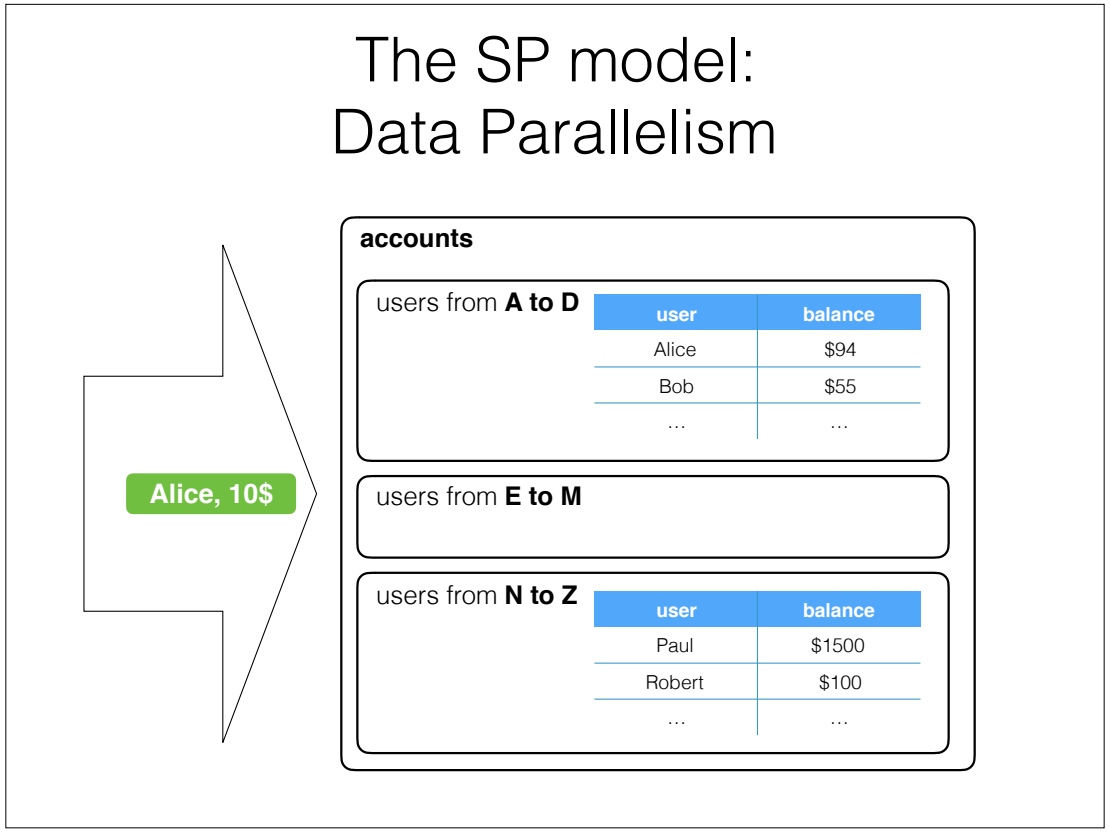
The SP Model



The SP Model: Task Parallelism



Nodes can be deployed on different machines and transfer data over the network.

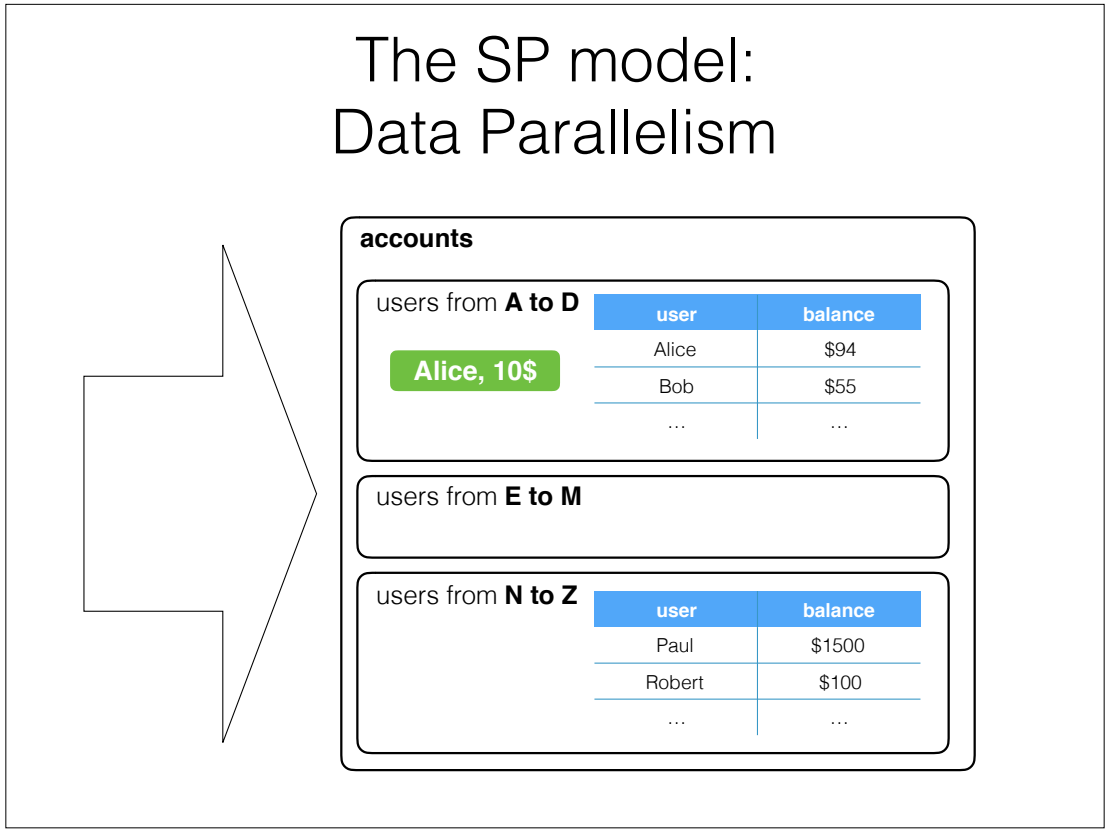


Stream elements can be partitioned in independent sets.

Every operator can be deployed more than once.

Each instance of the operator will process a partition of the elements.

The function that maps an element to a partition can be user-defined.

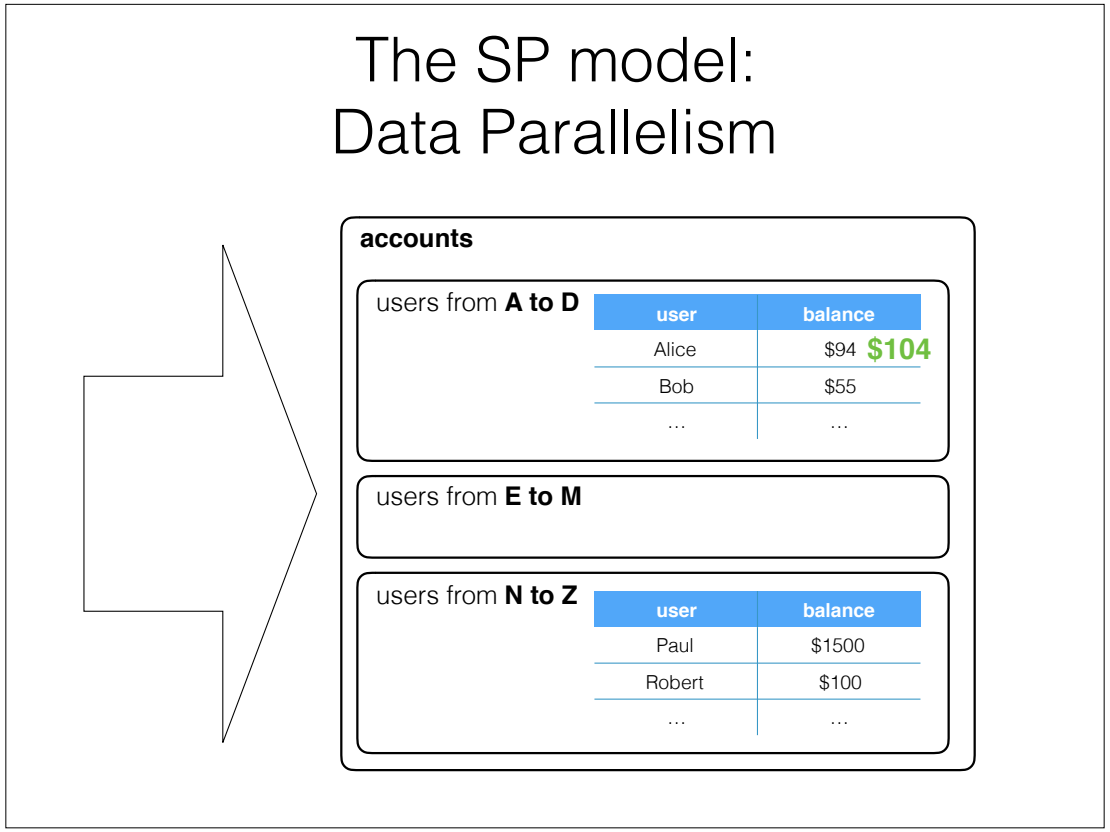


Stream elements can be partitioned in independent sets.

Every operator can be deployed more than once.

Each instance of the operator will process a partition of the elements.

The function that maps an element to a partition can be user-defined.

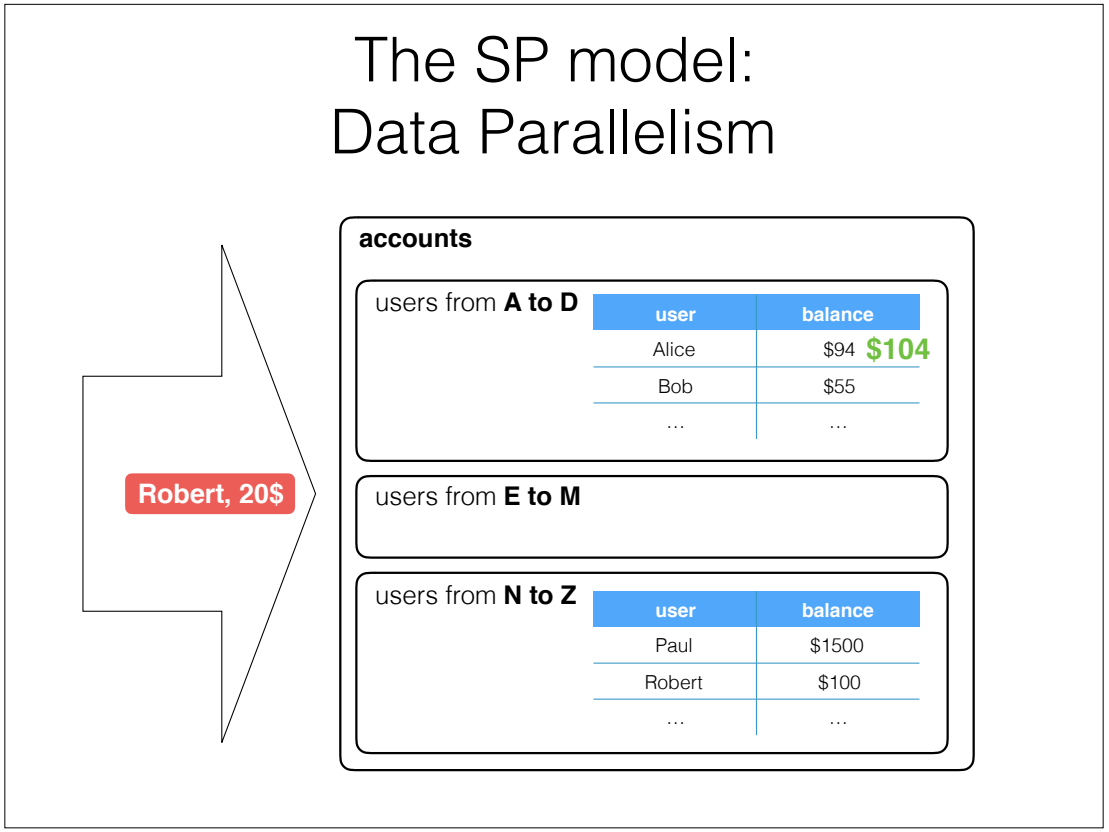


Stream elements can be partitioned in independent sets.

Every operator can be deployed more than once.

Each instance of the operator will process a partition of the elements.

The function that maps an element to a partition can be user-defined.

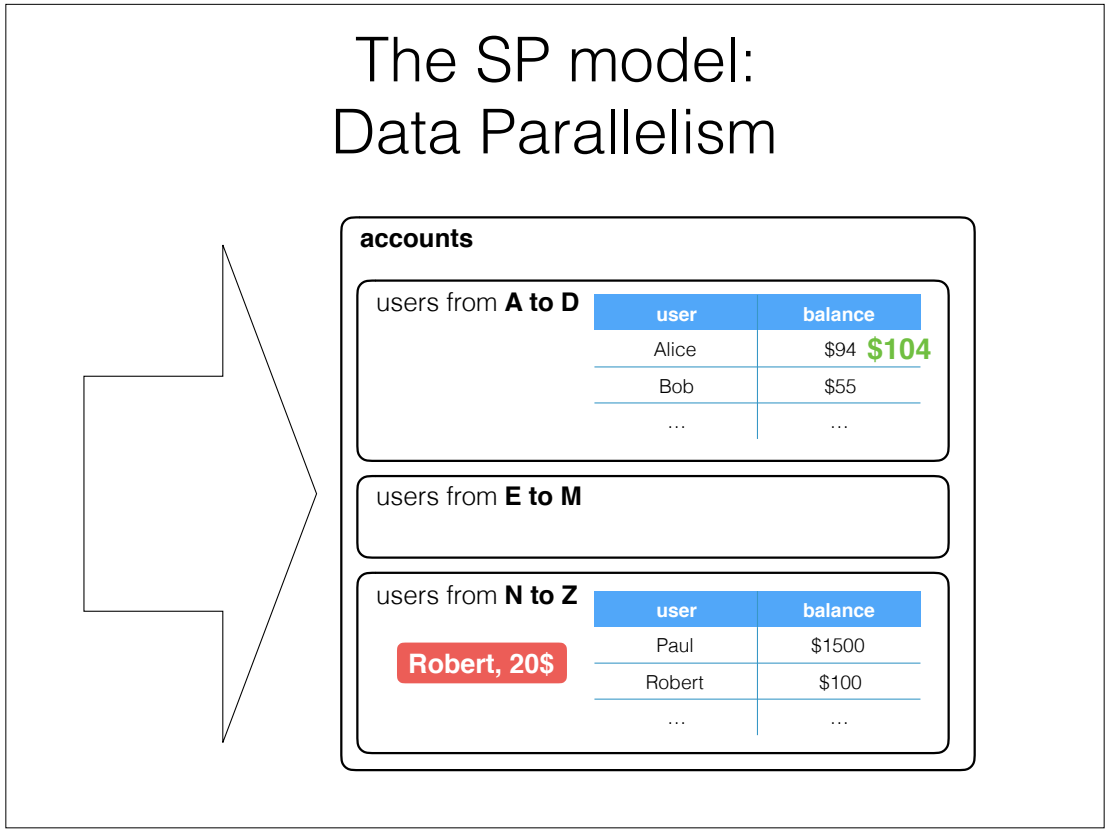


Stream elements can be partitioned in independent sets.

Every operator can be deployed more than once.

Each instance of the operator will process a partition of the elements.

The function that maps an element to a partition can be user-defined.

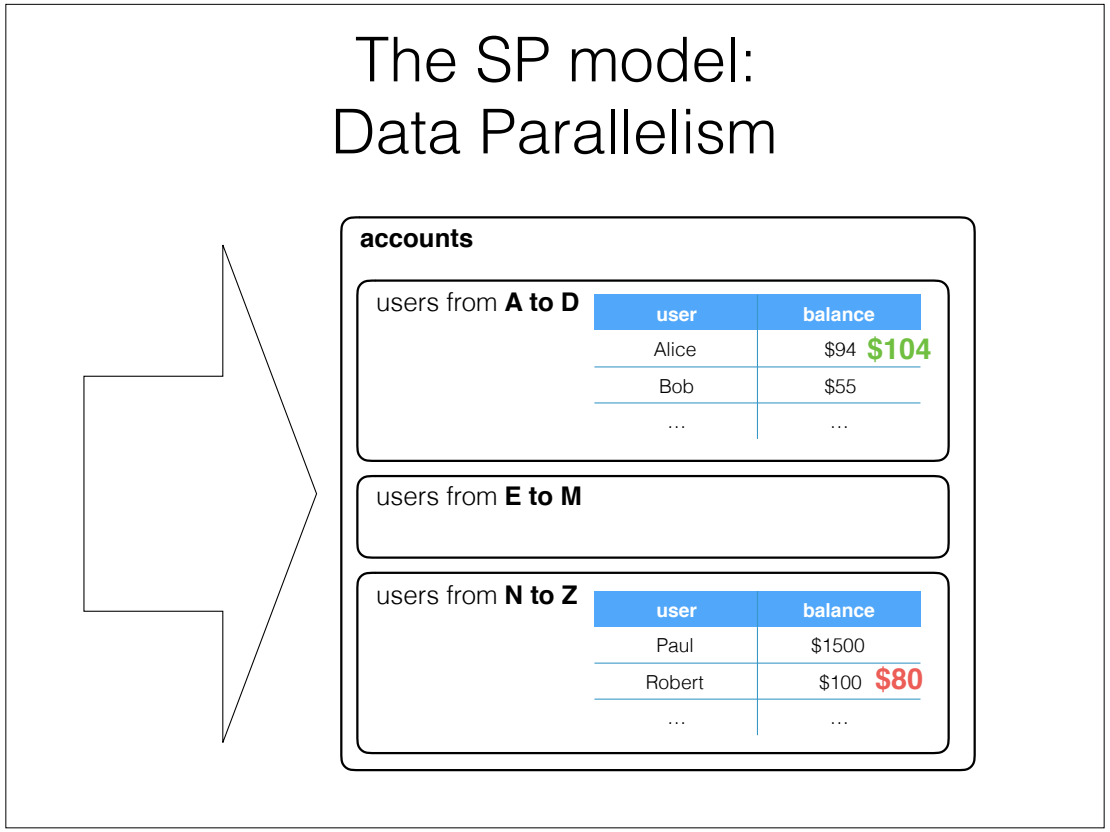


Stream elements can be partitioned in independent sets.

Every operator can be deployed more than once.

Each instance of the operator will process a partition of the elements.

The function that maps an element to a partition can be user-defined.

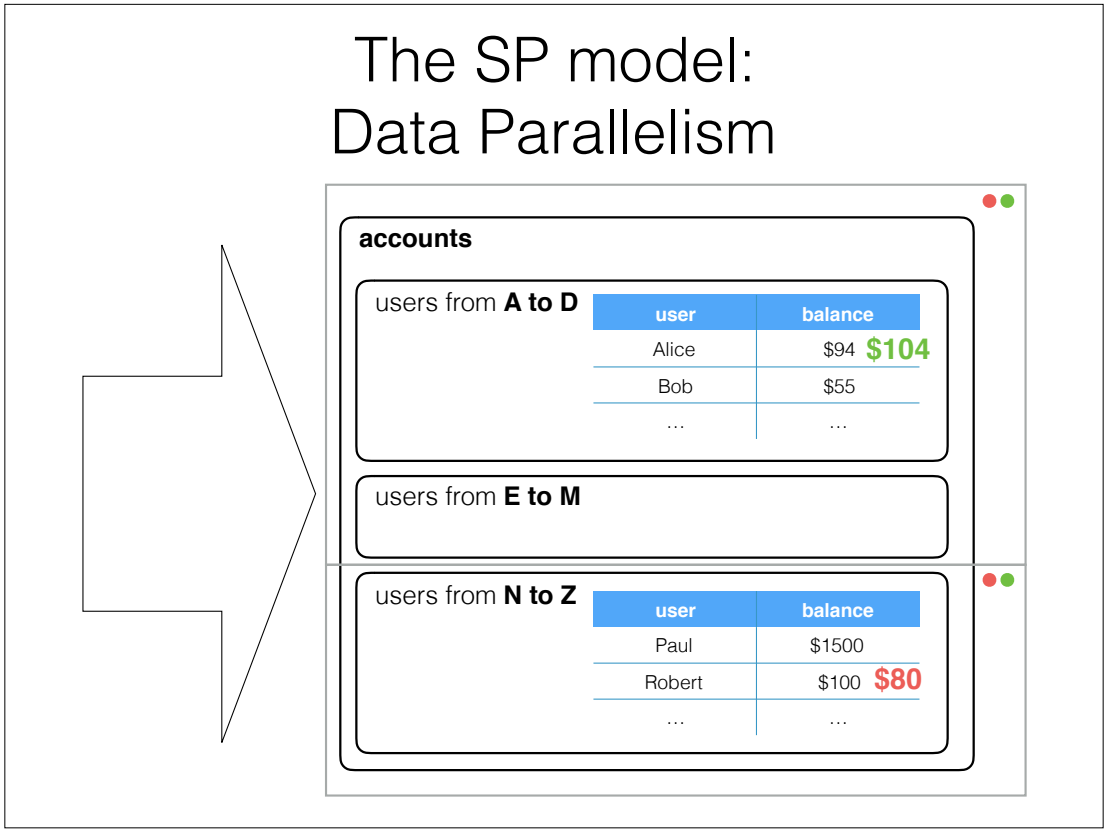


Stream elements can be partitioned in independent sets.

Every operator can be deployed more than once.

Each instance of the operator will process a partition of the elements.

The function that maps an element to a partition can be user-defined.



Stream elements can be partitioned in independent sets.

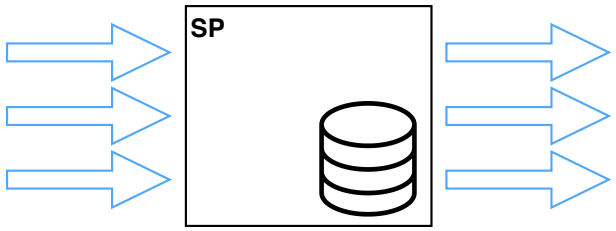
Every operator can be deployed more than once.

Each instance of the operator will process a partition of the elements.

The function that maps an element to a partition can be user-defined.

The Solution

- Built for **distributed computation** ✓
- Exposes **queryable state**
- Offers **transactional guarantees** on the state



Task and data parallelism enable distributed computation.

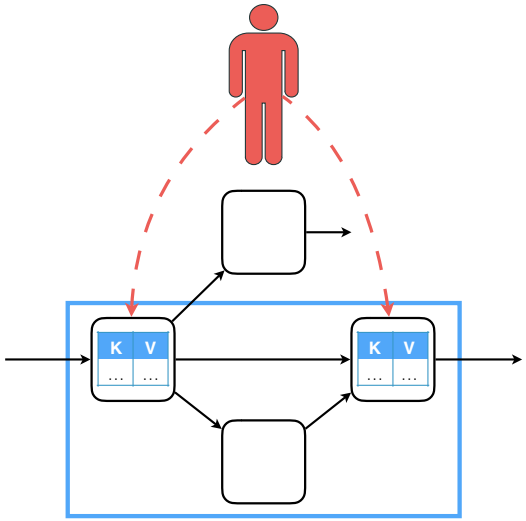
The rest of the presentation will investigate queryable state and transactional guarantees.

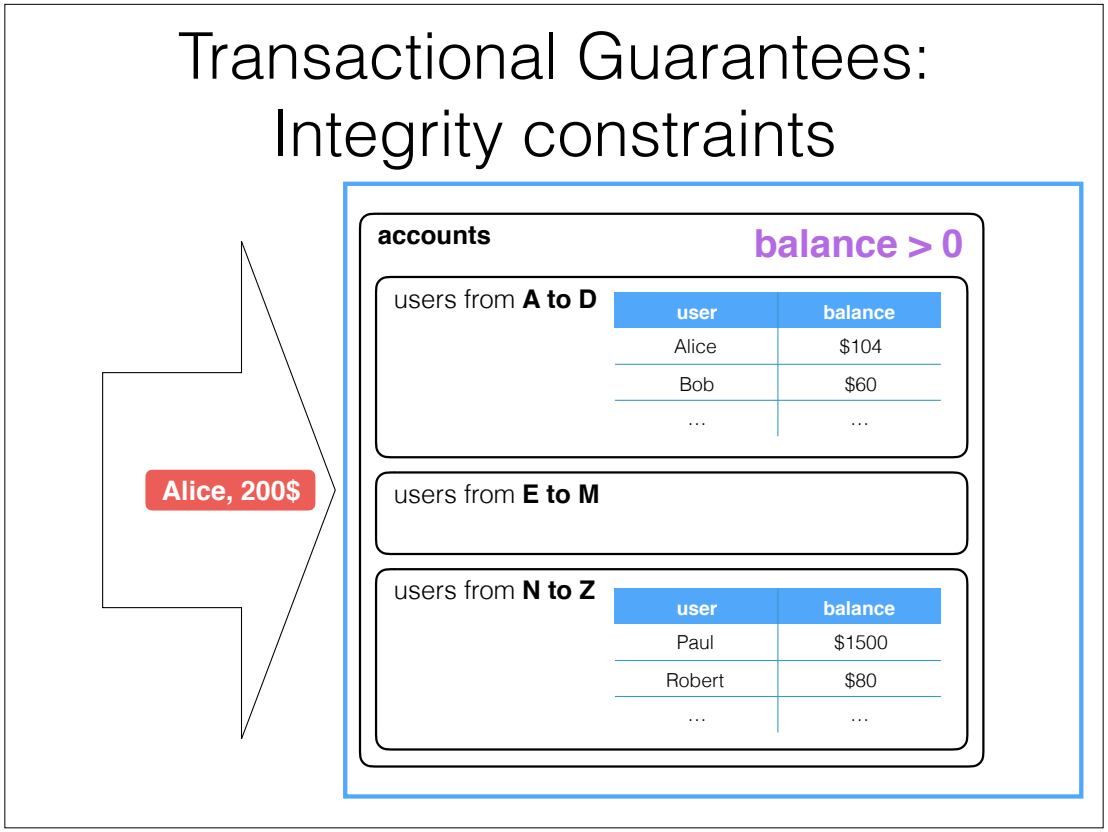
The Transactional Subgraph

A **transactional subgraph** is a subset of the original graph of computation

We enforce **transactional guarantees** both on **read** and on **update** of the internal state of state operators

State operators in a transactional subgraph **expose their internal state**

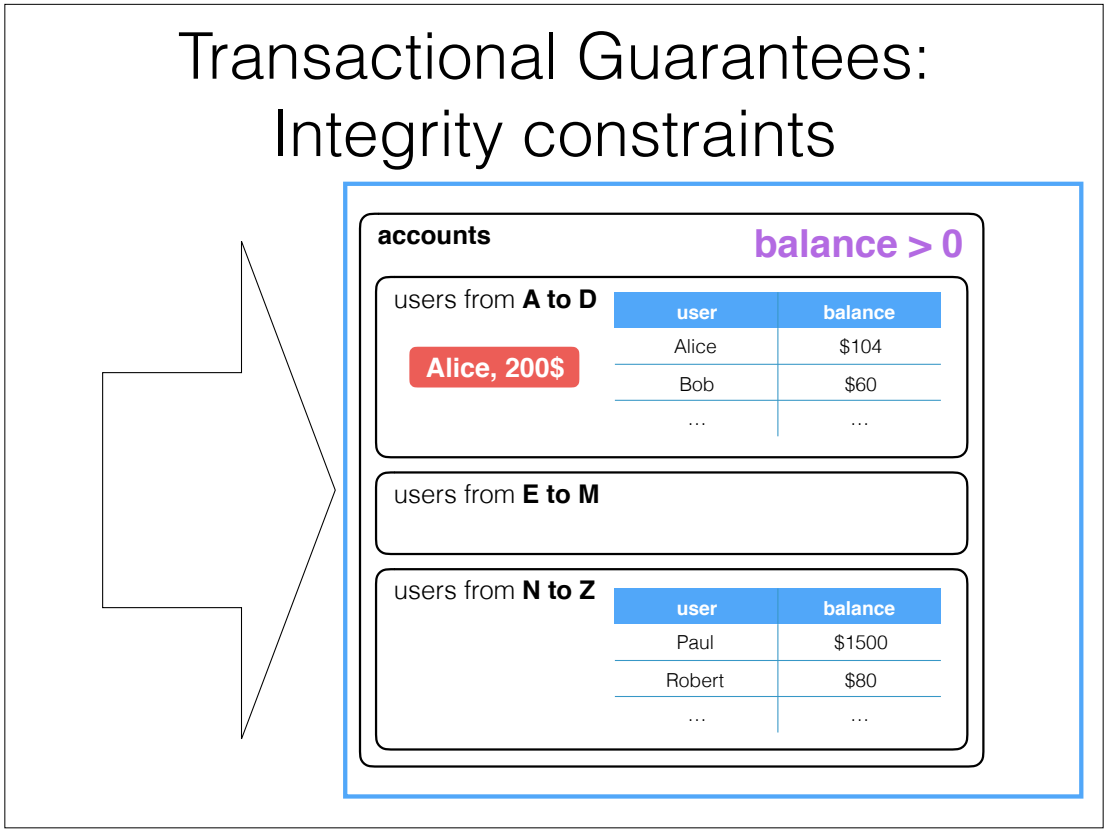




We give the possibility to specify per-state-operator integrity constraints.

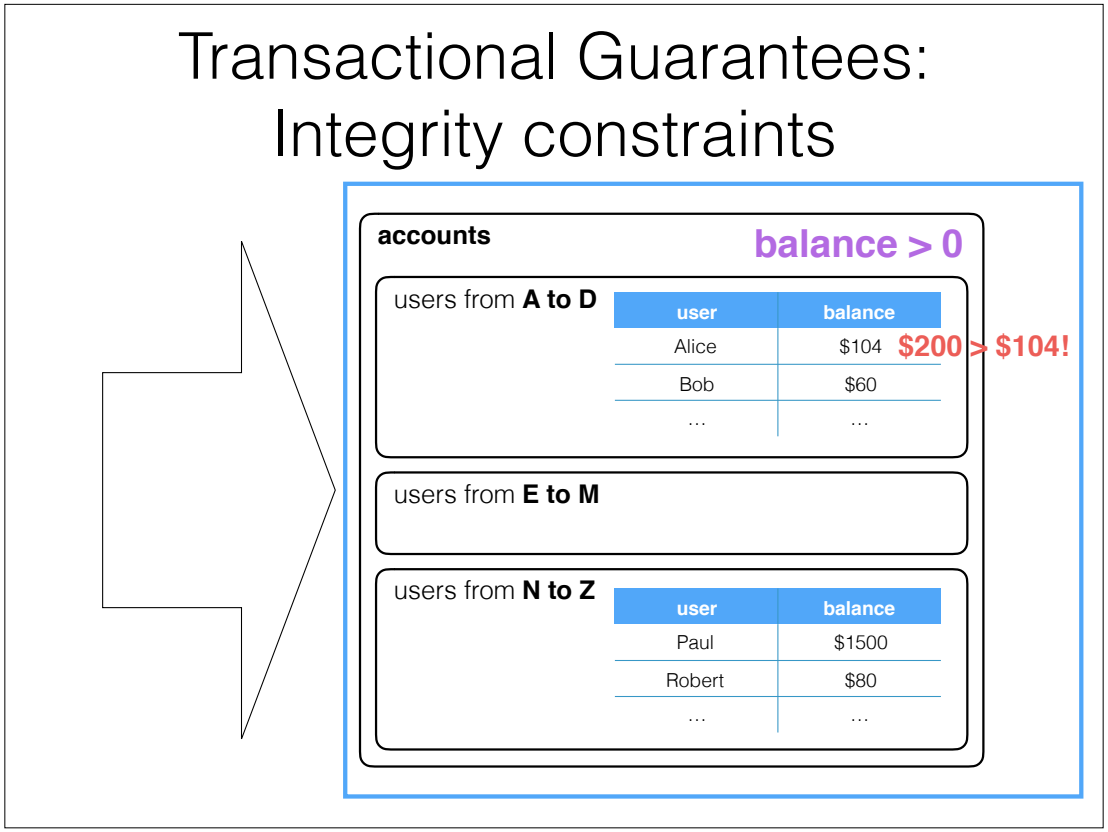
If a constraint is violated the operation is considered not valid and the state will not be affected.

Transactional Guarantees: Integrity constraints



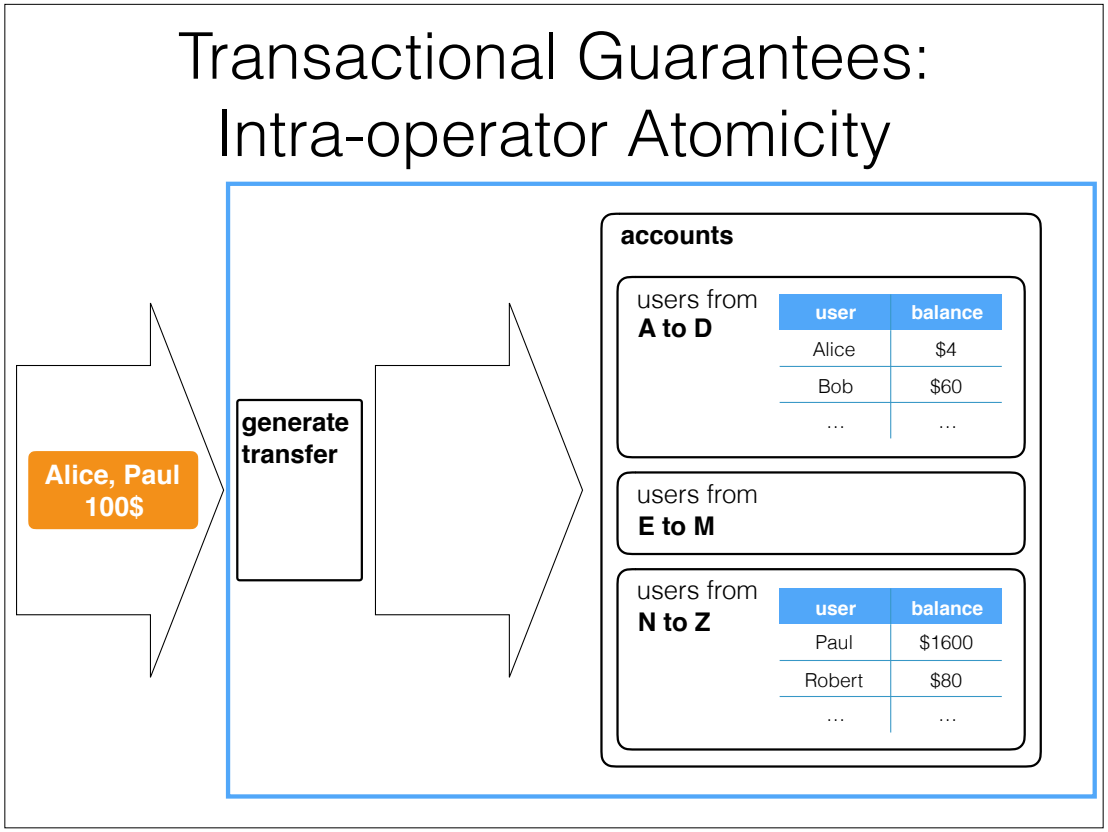
We give the possibility to specify per-state-operator integrity constraints.

If a constraint is violated the operation is considered not valid and the state will not be affected.



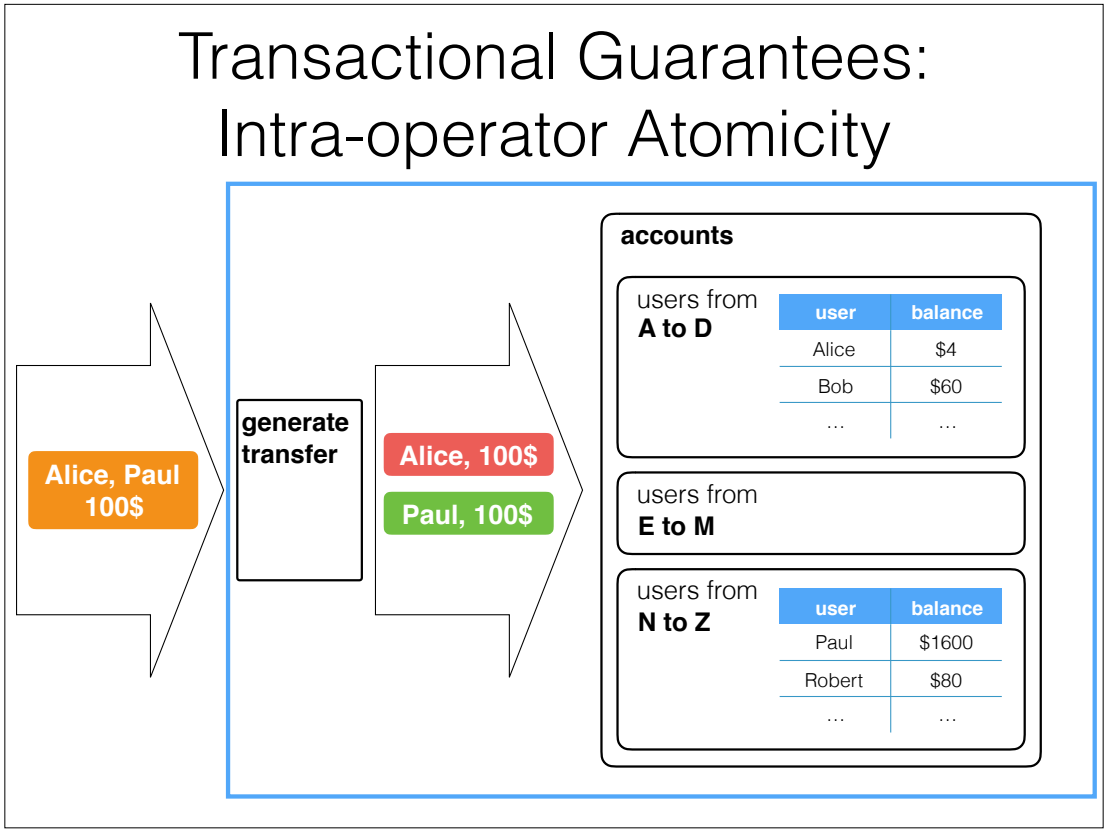
We give the possibility to specify per-state-operator integrity constraints.

If a constraint is violated the operation is considered not valid and the state will not be affected.



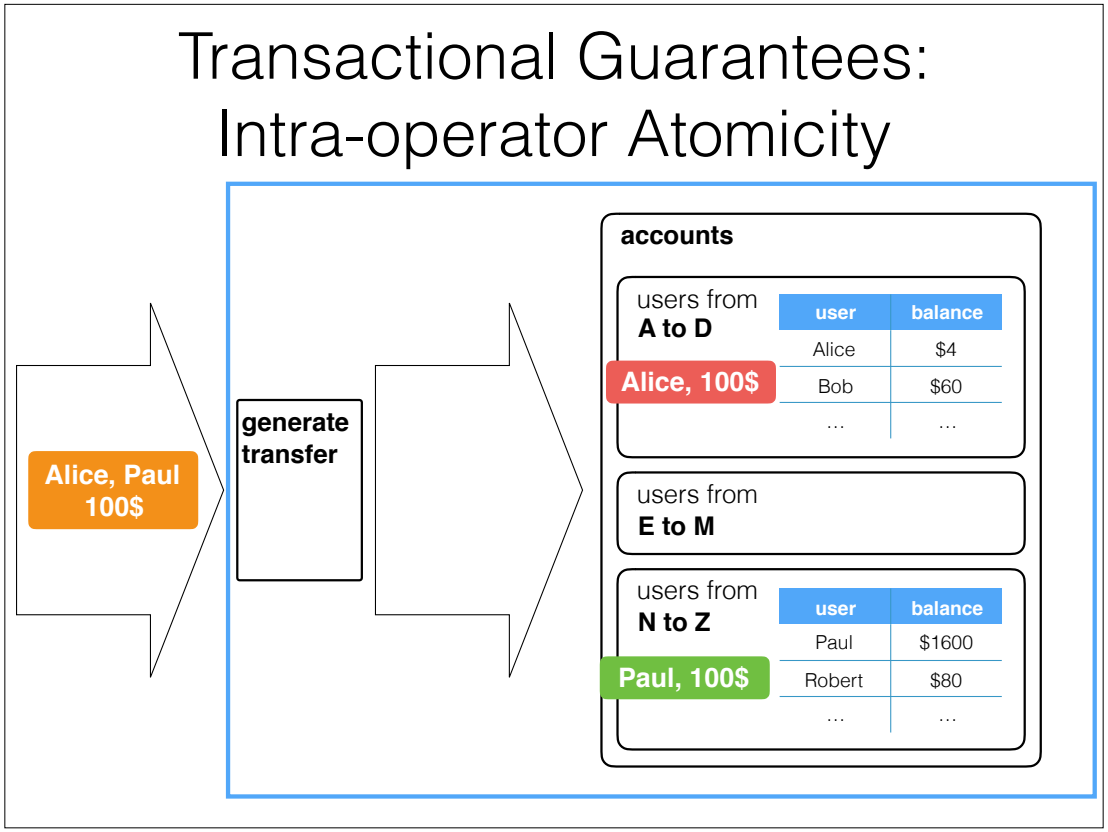
If a transaction affects more than one partition of the same operator and aborts (fails) on one, than it will abort on every partition.

No state will be affected.



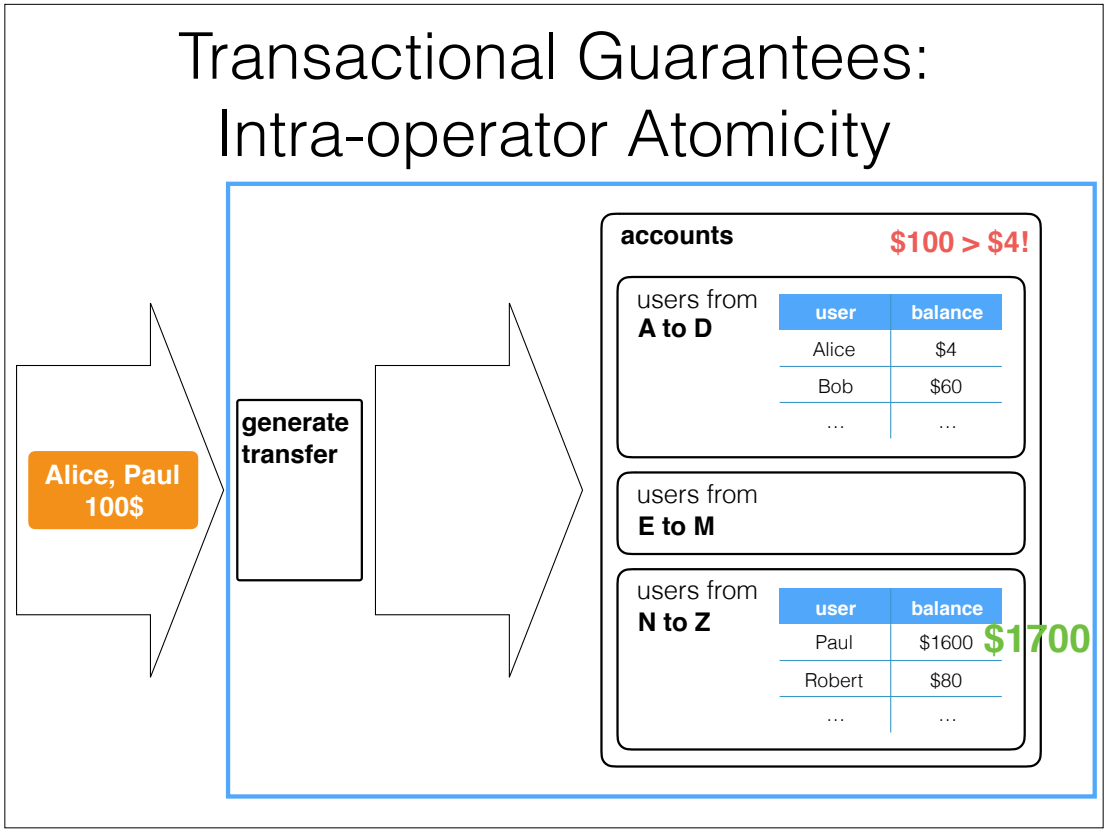
If a transaction affects more than one partition of the same operator and aborts (fails) on one, than it will abort on every partition.

No state will be affected.



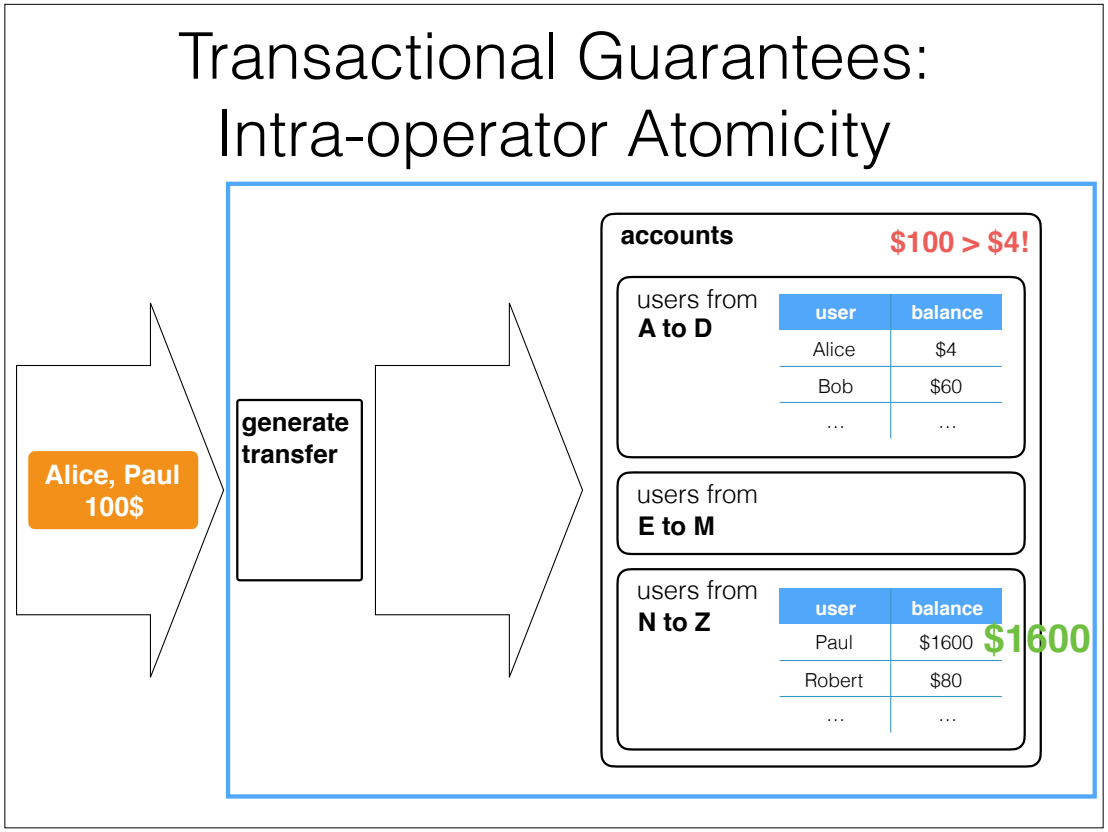
If a transaction affects more than one partition of the same operator and aborts (fails) on one, than it will abort on every partition.

No state will be affected.



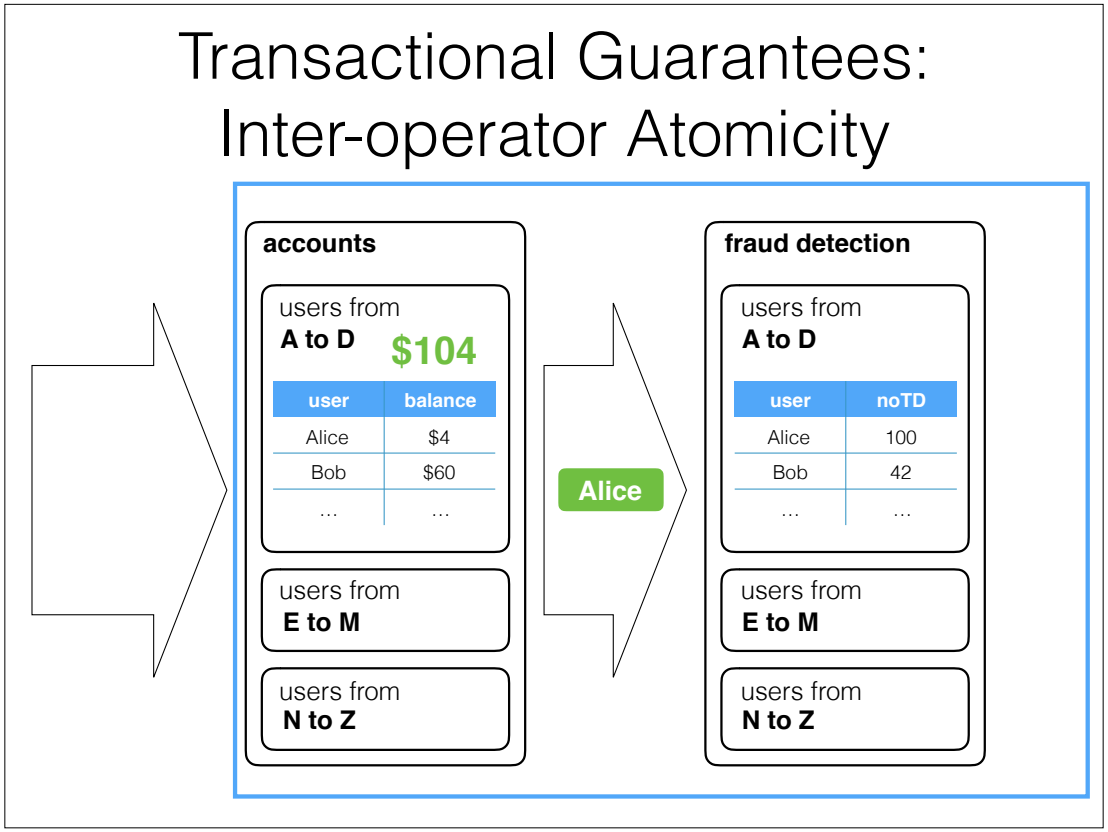
If a transaction affects more than one partition of the same operator and aborts (fails) on one, than it will abort on every partition.

No state will be affected.



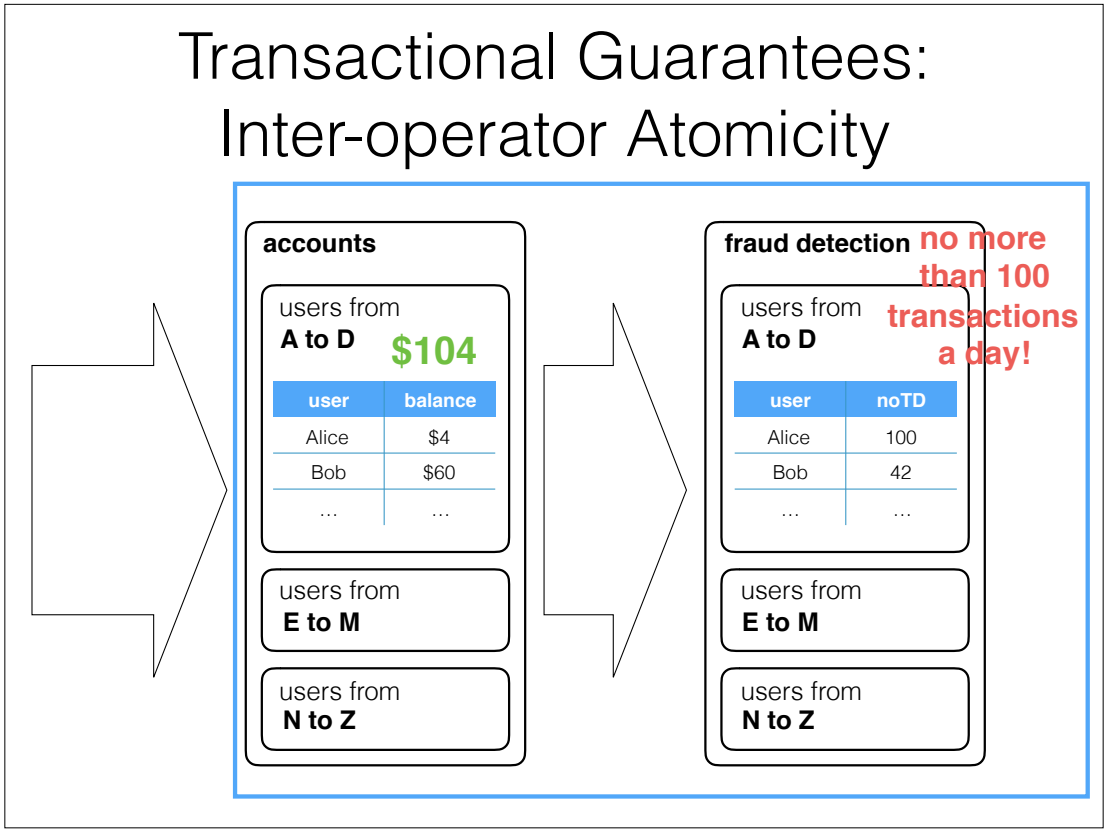
If a transaction affects more than one partition of the same operator and aborts (fails) on one, than it will abort on every partition.

No state will be affected.



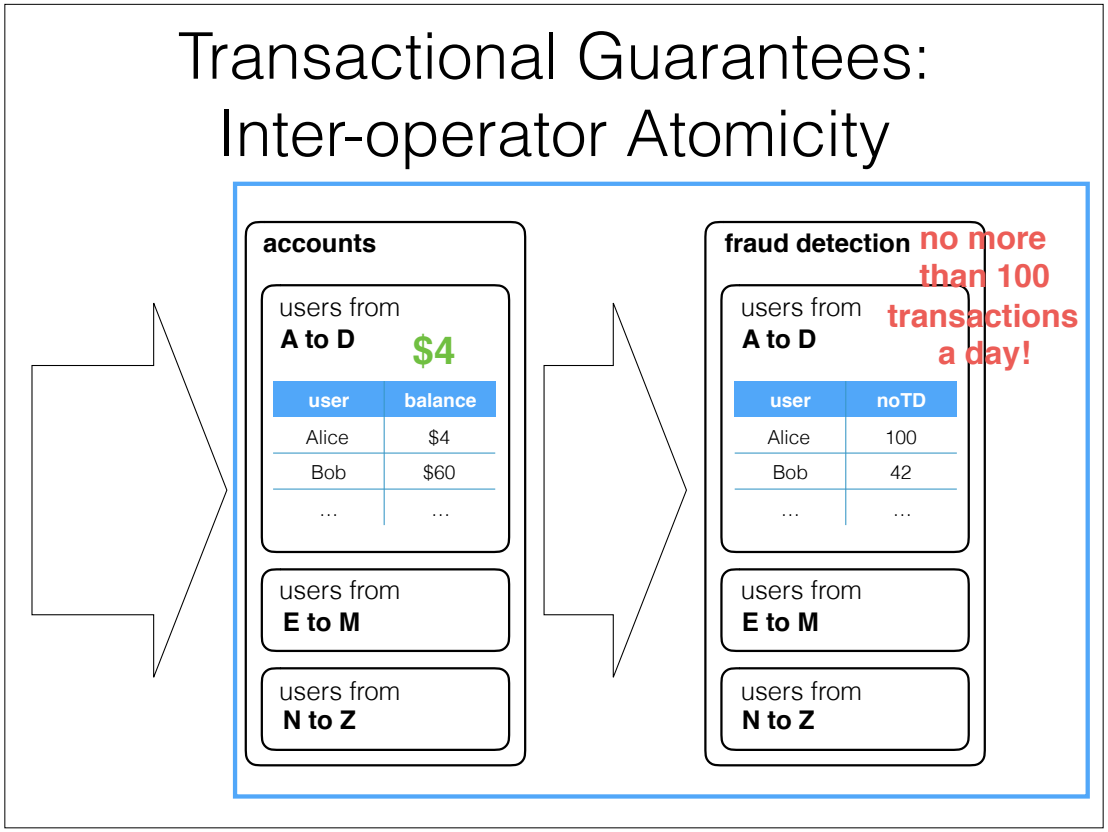
If a transaction affects more than one state operator in the same transactional subgraph and aborts on one, than it will abort on every operator.

No state will be affected.



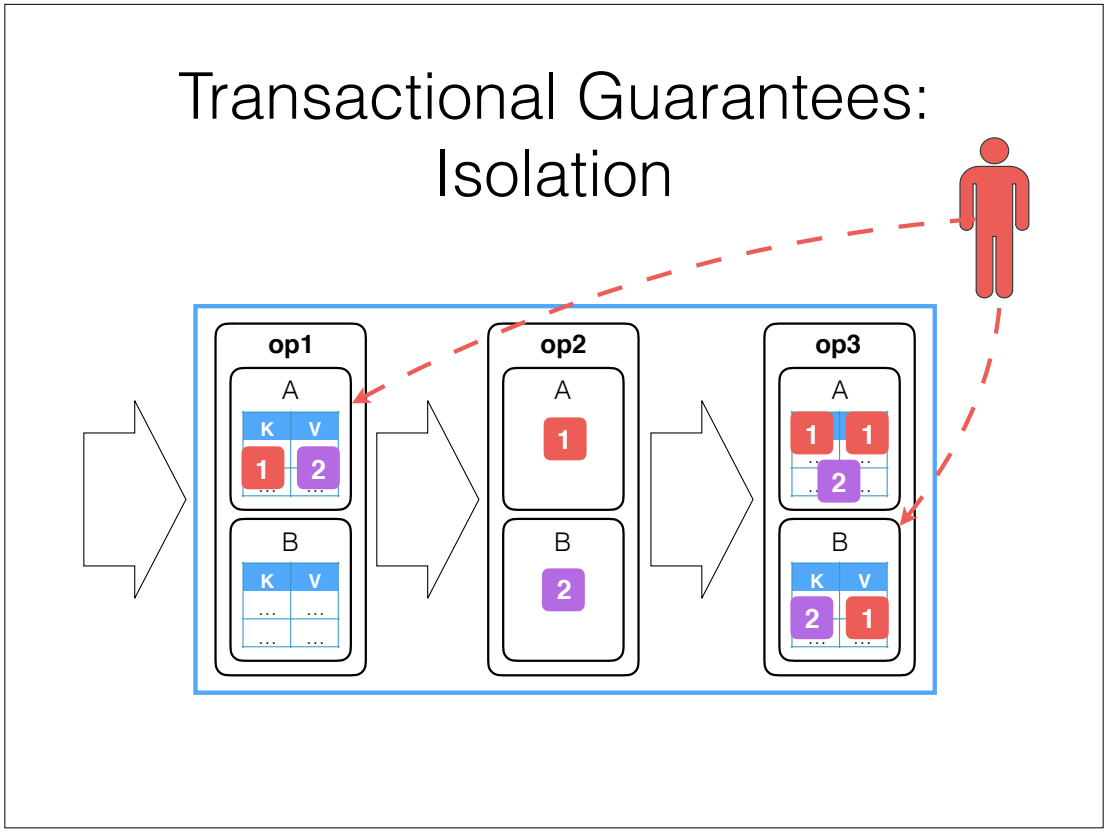
If a transaction affects more than one state operator in the same transactional subgraph and aborts on one, than it will abort on every operator.

No state will be affected.



If a transaction affects more than one state operator in the same transactional subgraph and aborts on one, than it will abort on every operator.

No state will be affected.



There can be various possible interleaving of operations in different operators and in different partitions.

Interleaving generates conflicts among operations.

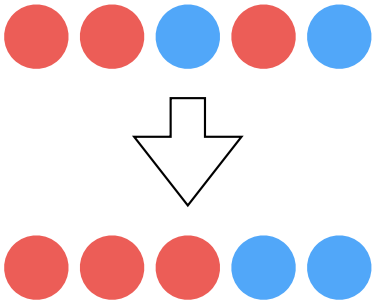
Handling or not some type of conflicts leads to different levels of isolation.

Transactional Guarantees: Isolation

Depending on the implementation of state operators and transaction coordination we can achieve **different levels of isolation**

Our model support **serializable** level of isolation

Transactions produce the same effect as some **serial execution**



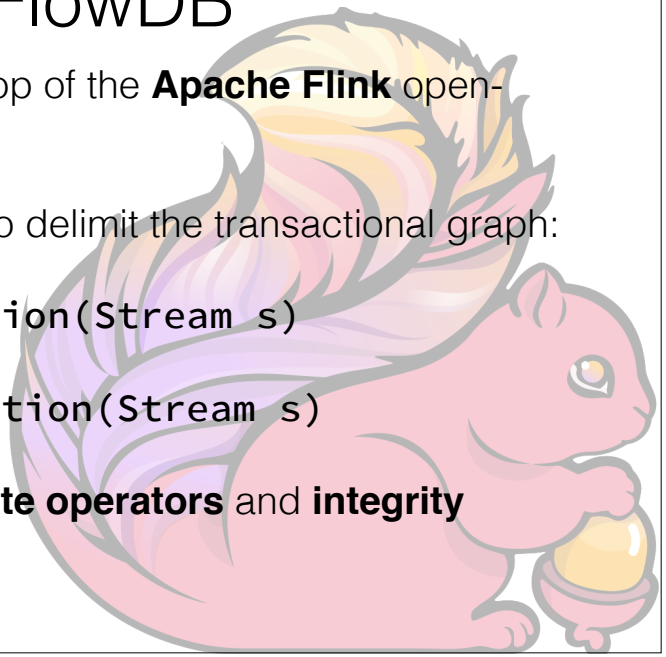
The Implementation: FlowDB

- Implemented on top of the **Apache Flink** open-source project
- We provide APIs to delimit the transactional graph:

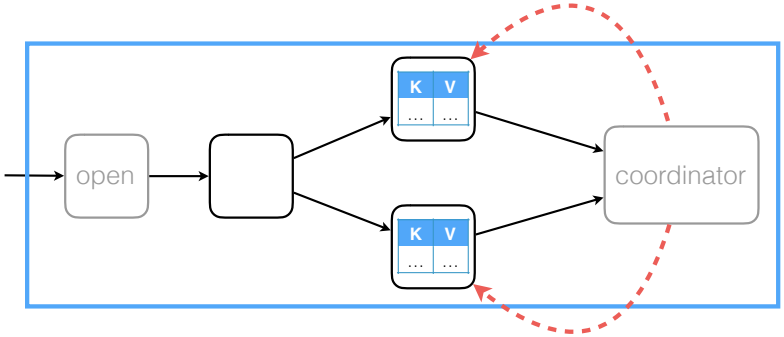
```
openTransaction(Stream s)

closeTransaction(Stream s)
```

- And to specify **state operators** and **integrity constraints**



Implementation: Transaction handling



Open enriches elements with a unique **transaction ID** for tracking purpose

The c*oordinator* is responsible for **merging the results** of the integrity checks and feed them back to state operators

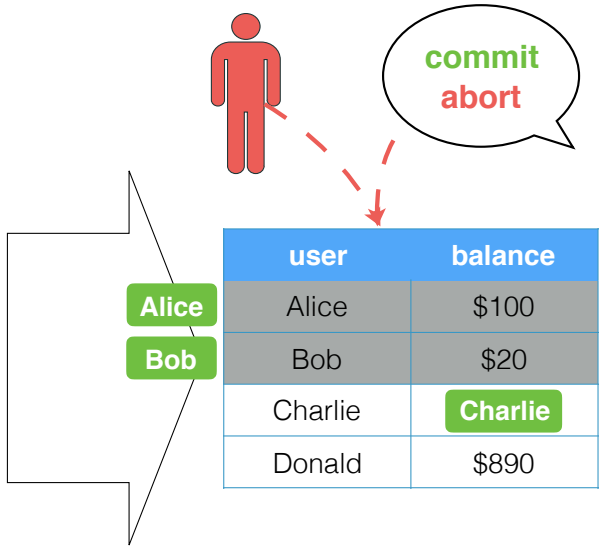
Implementation: State Operators

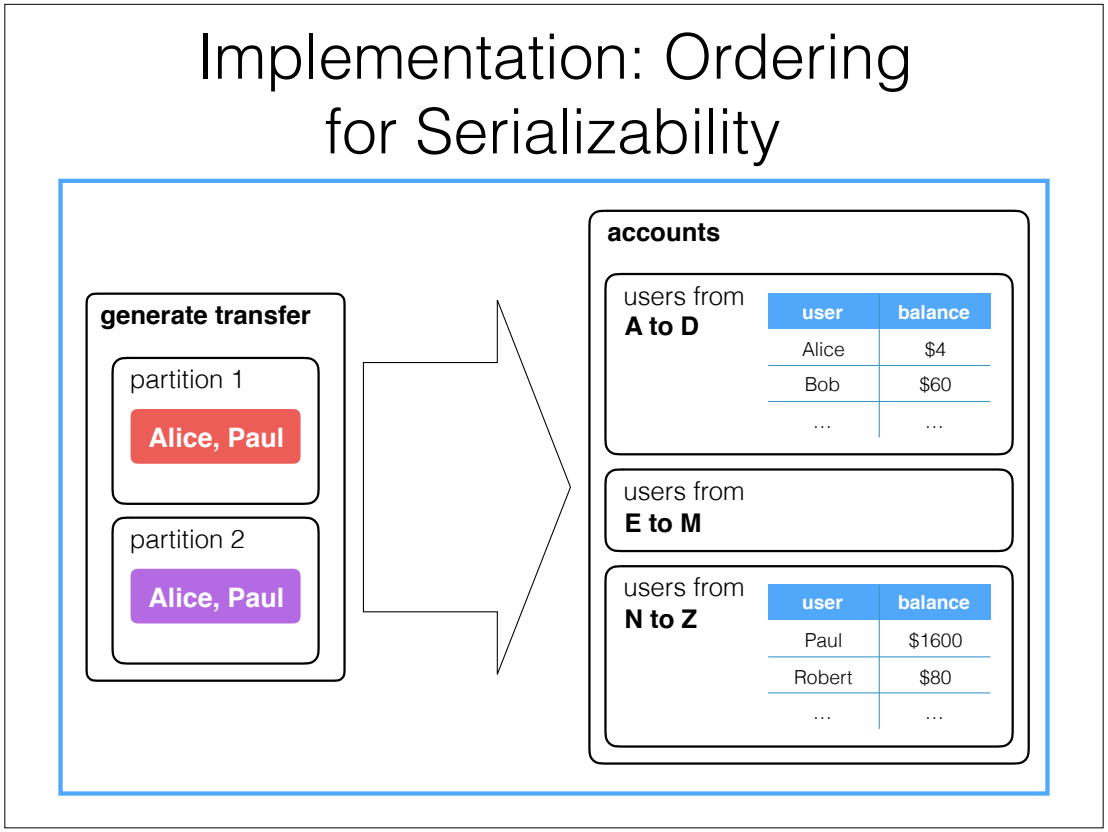
We use **key-level locking** to provide isolation

If one key is locked, records with the same key are **enqueued until unlock**

Resources are unlocked on coordinator notification

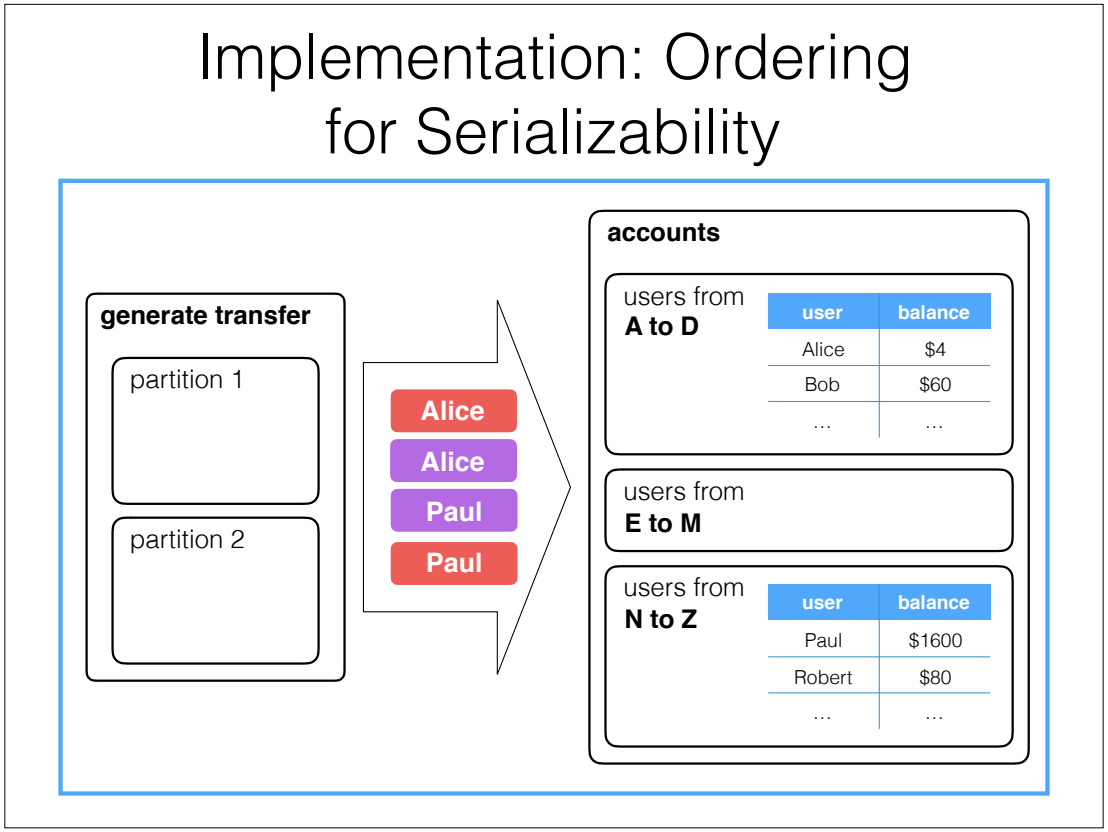
Read-only transactions use the same locking mechanism





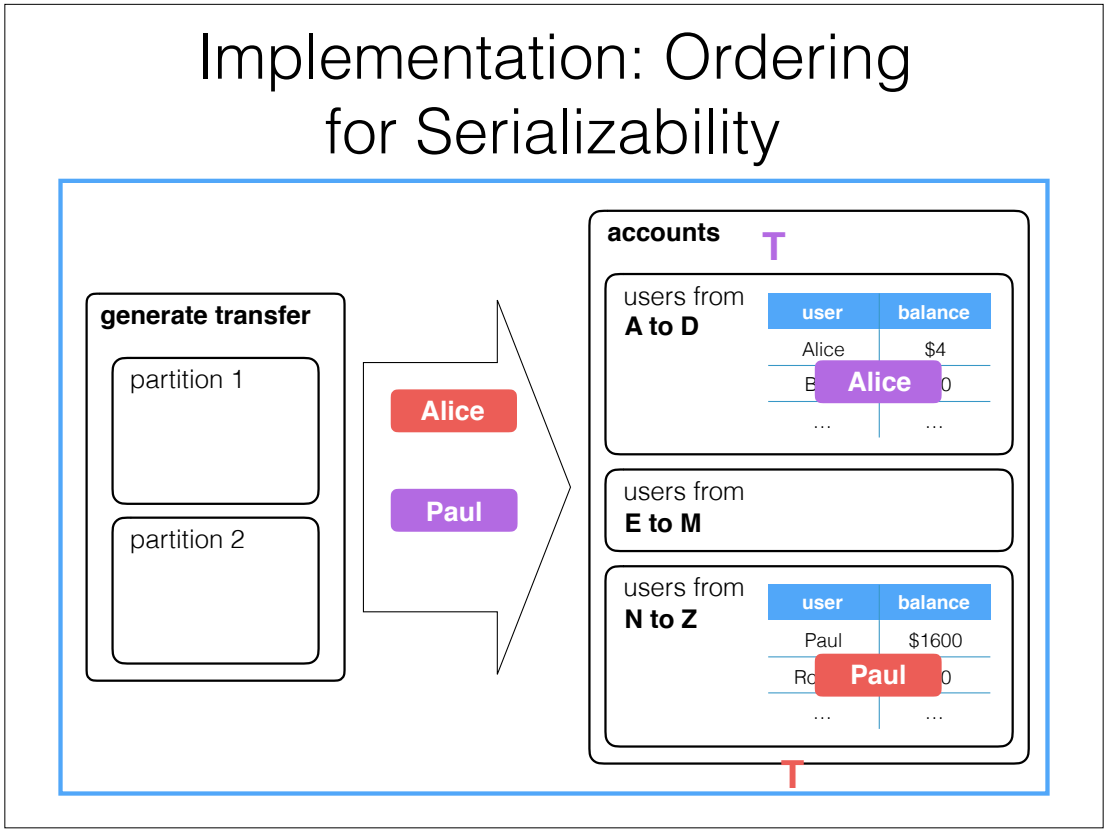
The graph of computation doesn't provide us with any guarantee about the ordering of record processing by downstream operators.

It could be that two transactions affect some state in different order.



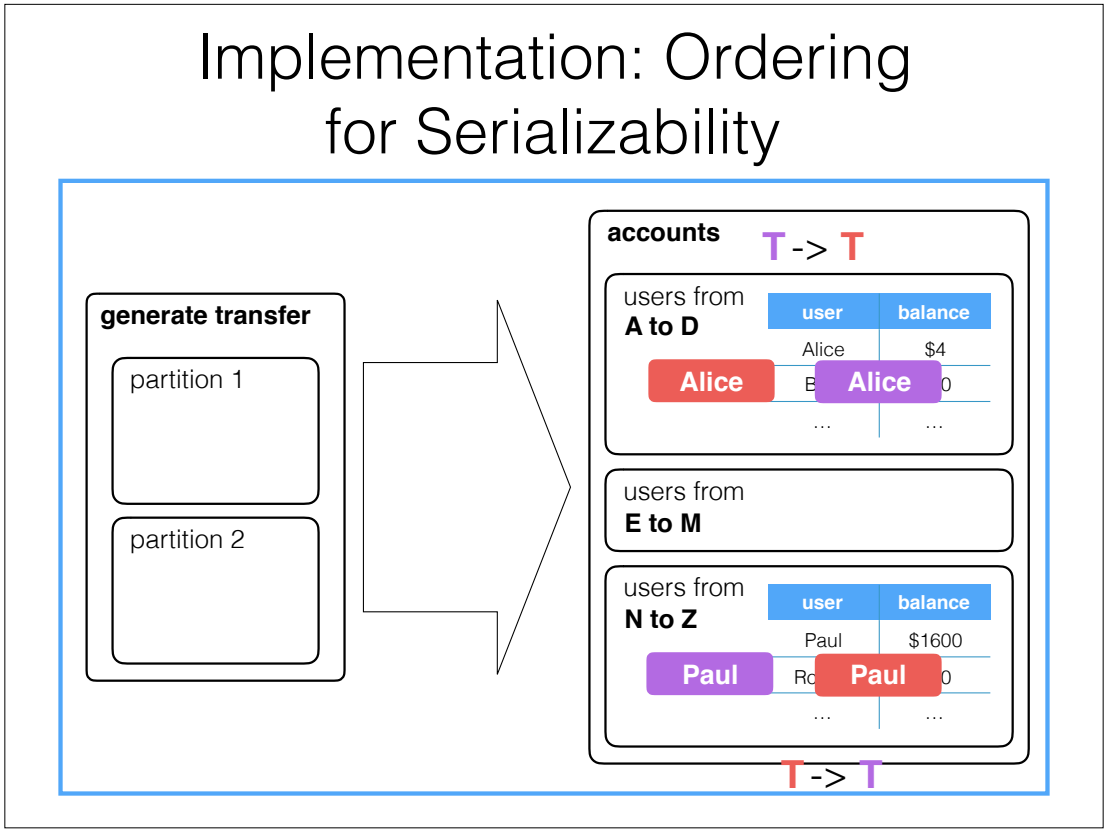
The graph of computation doesn't provide us with any guarantee about the ordering of record processing by downstream operators.

It could be that two transactions affect some state in different order.



The graph of computation doesn't provide us with any guarantee about the ordering of record processing by downstream operators.

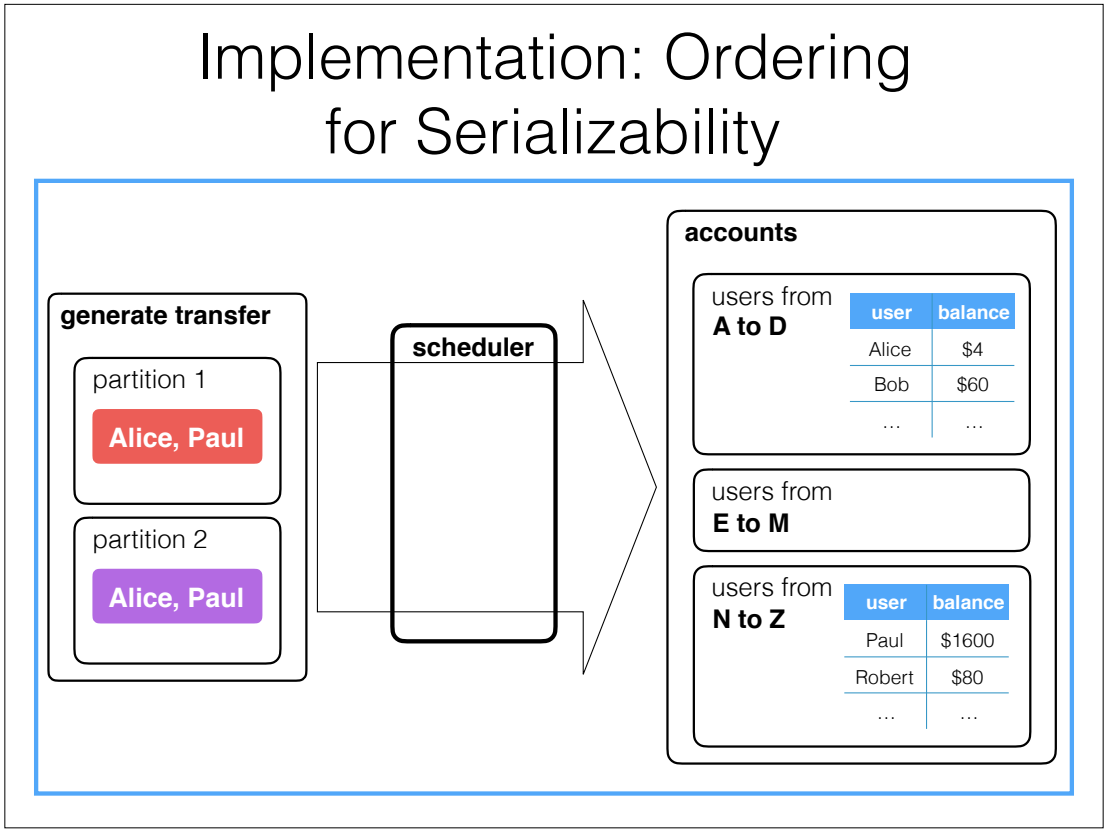
It could be that two transactions affect some state in different order.



The graph of computation doesn't provide us with any guarantee about the ordering of record processing by downstream operators.

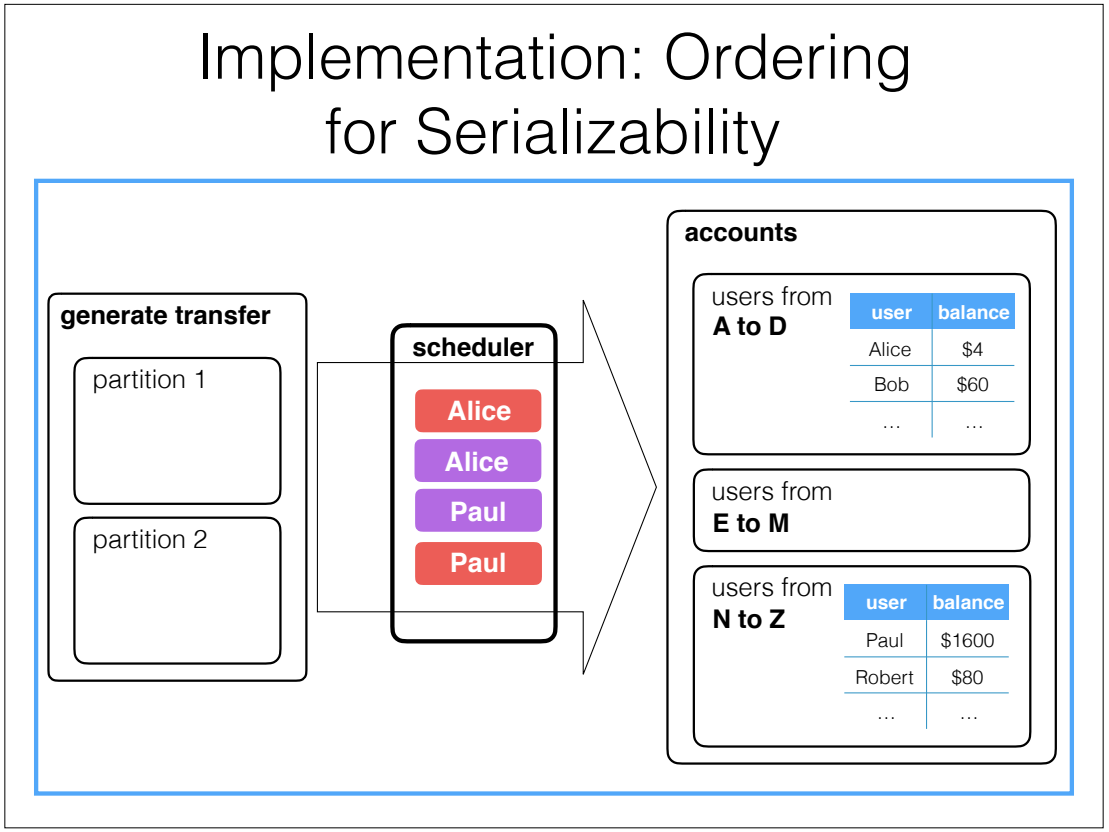
It could be that two transactions affect some state in different order.

Implementation: Ordering for Serializability



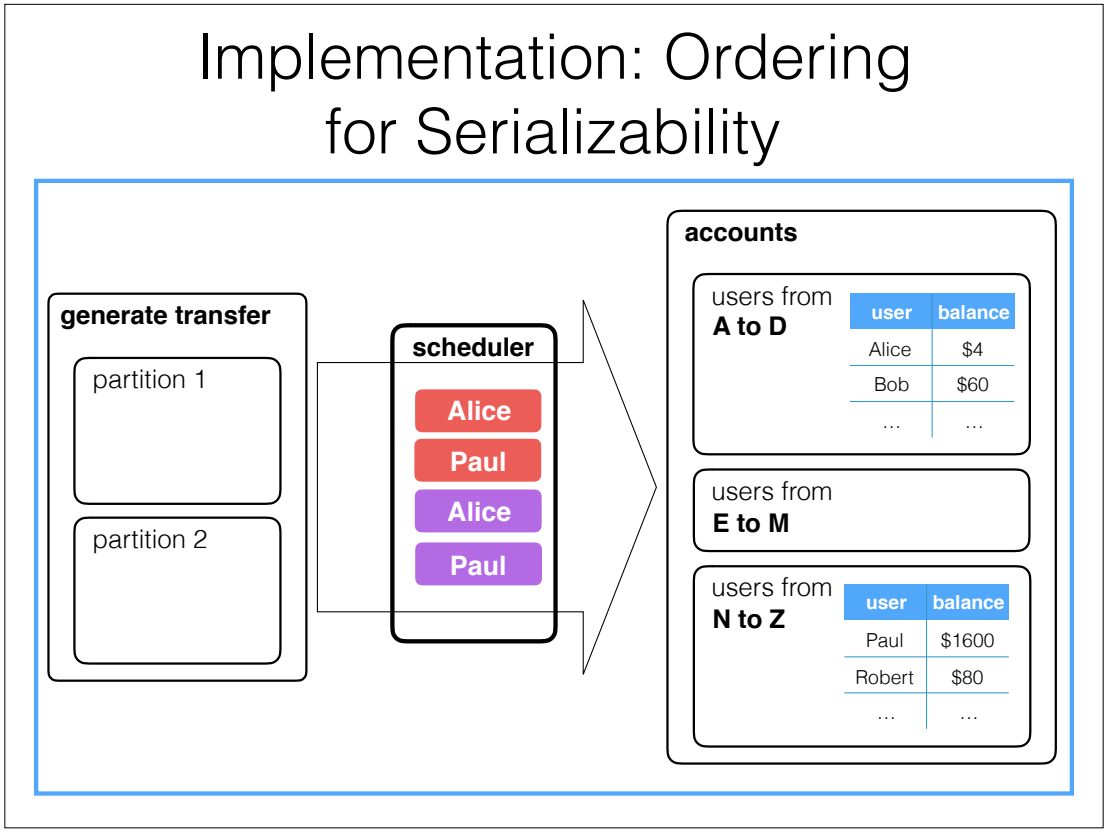
That's why we need *schedulers* to impose a total order on elements.

Implementation: Ordering for Serializability



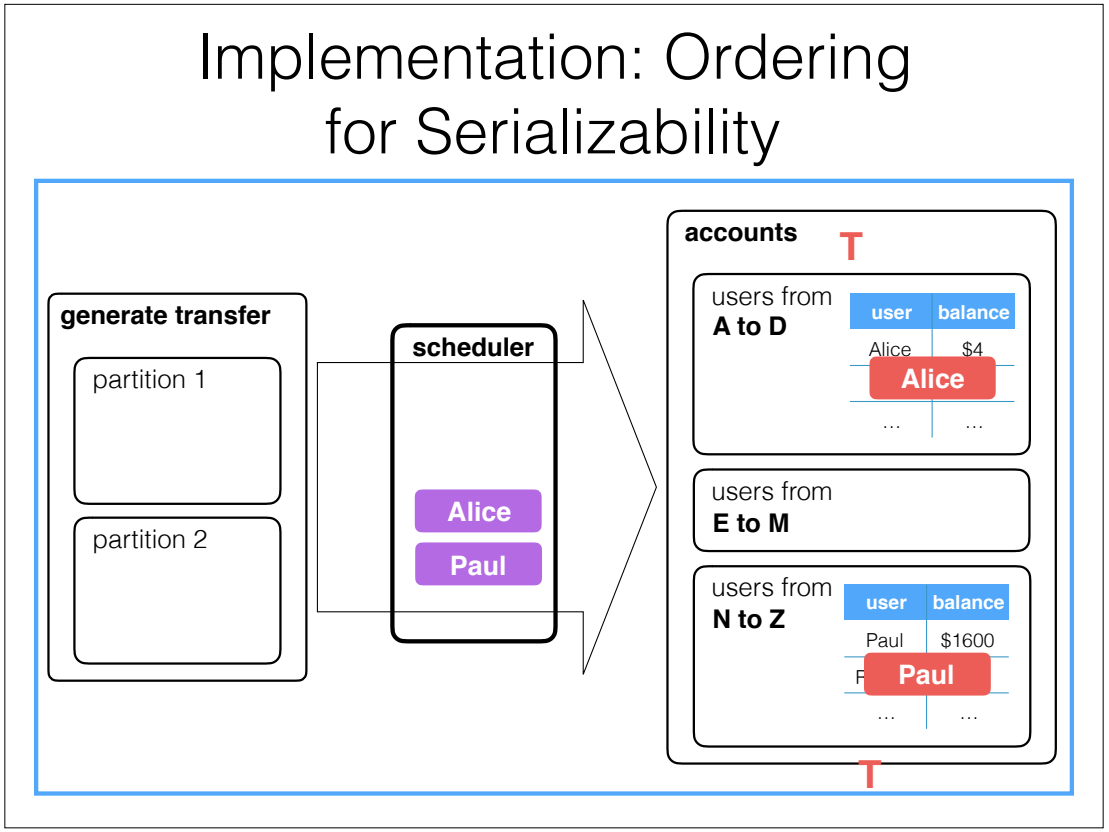
That's why we need *schedulers* to impose a total order on elements.

Implementation: Ordering for Serializability



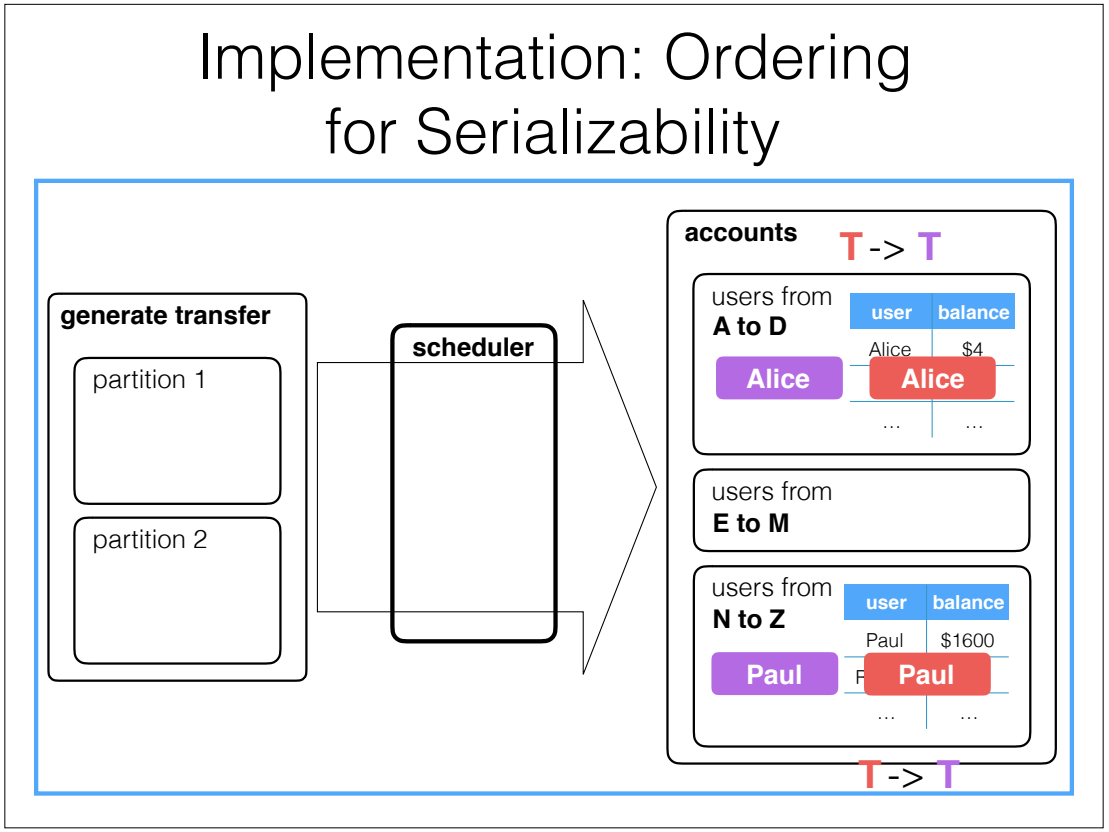
That's why we need *schedulers* to impose a total order on elements.

Implementation: Ordering for Serializability



That's why we need *schedulers* to impose a total order on elements.

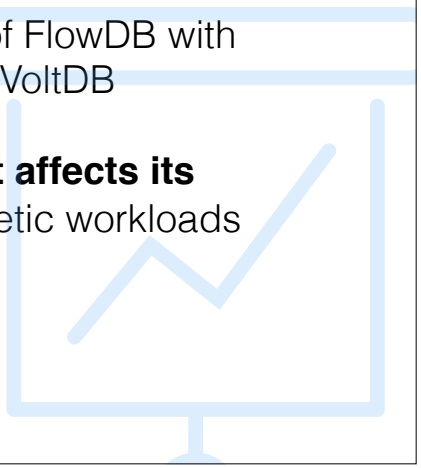
Implementation: Ordering for Serializability



That's why we need *schedulers* to impose a total order on elements.

The Evaluation

- Compare the performance of FlowDB with a **state-of-the-art solution**, VoltDB
- Identify the **parameters that affects its performance** through synthetic workloads



The Evaluation: VoltDB

- It is a **distributed, in-memory** database
- It **partitions data** in shards
- It executes transactions as single-threaded stored procedures, which are **precompiled and optimized**
- If it is a **multi-partition** transaction, it requires **coordination**

Comparing FlowDB

Bank transfer example (no fraud detection mechanism).

100k accounts on 8 partitions, 200k transfers with **uniformly selected** accounts.

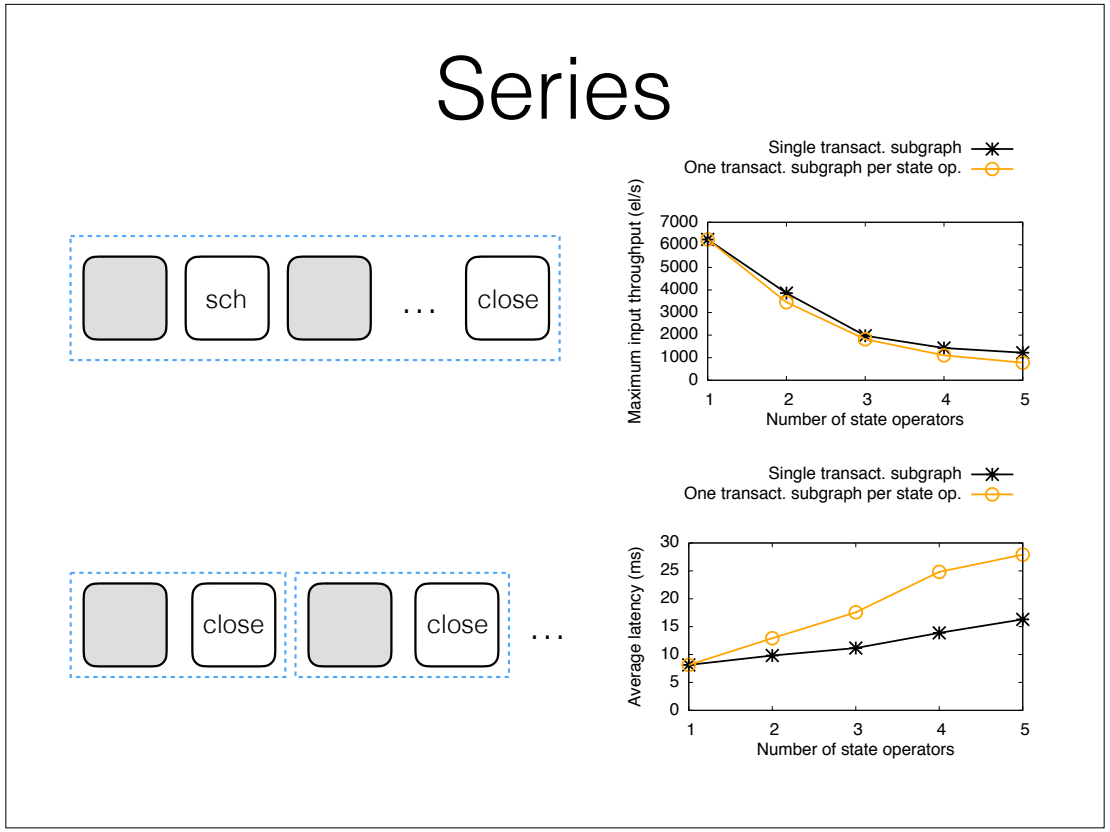
20 Amazon EC2 t2 XL instances, each equipped with 4 CPU cores and 16 GB of RAM.

	Avg latency	Throughput
Flink	3.1 ms	68705 t/s
FlowDB	8.2 ms	6235 tr/s
VoltDB	5092 ms	589 tr/s

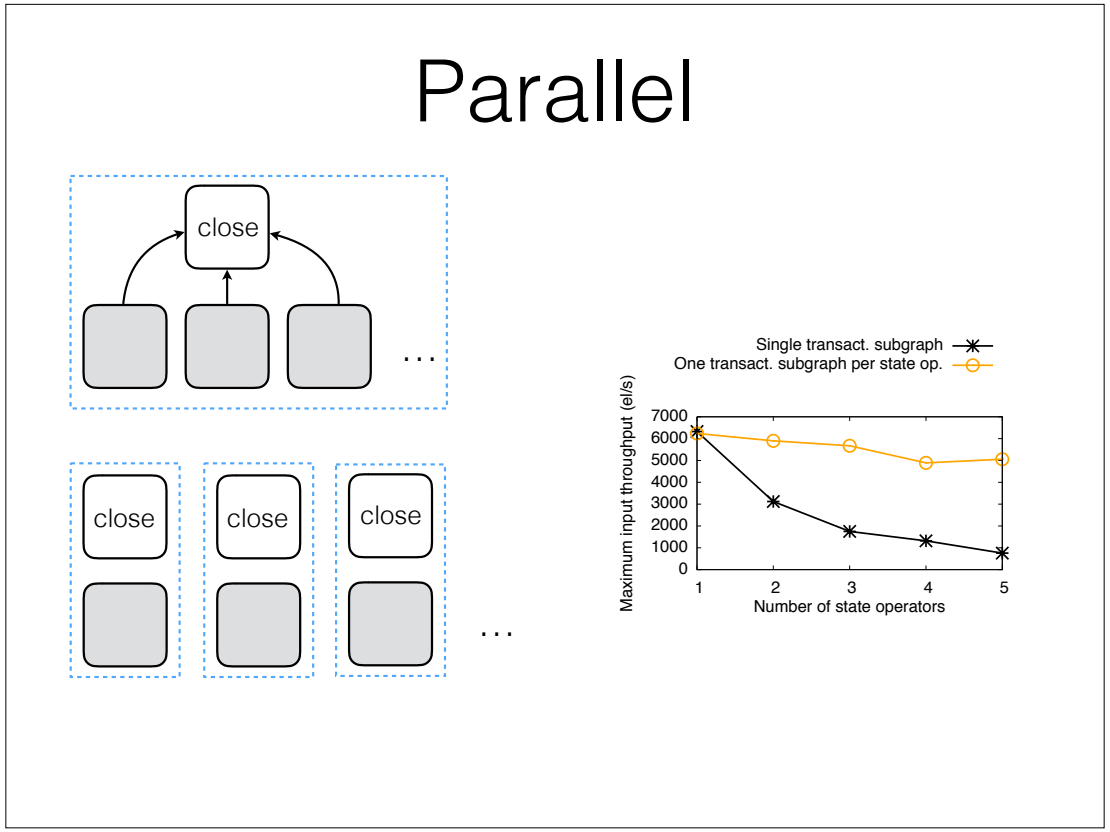
Please note that pure Flink **does not run the same application** as FlowDB.

We provide results only to show the overhead of providing transactional behavior.

VoltDB shows its b high frequency of multi-partition transactions.



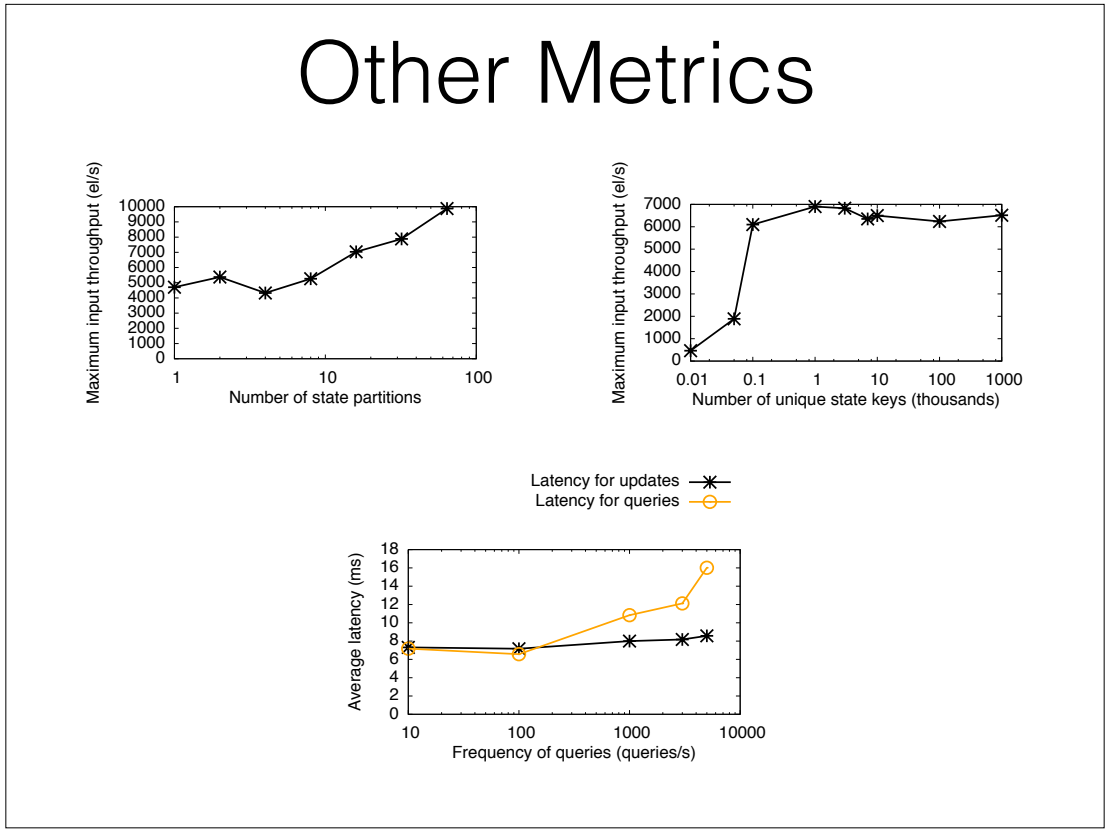
The results above let us conclude that in FlowDB the cost for ensuring isolation through a scheduler is negligible with respect to the cost of opening transactions, locking resources, and establishing the validity of a transaction.



In the case of a single transactional subgraph, the maximum throughput decreases with the number of state operators.

This is due to the increased volume of input data and to the need for collecting results from all the state operators to determine the overall validity of a transaction.

Conversely, in the case of multiple transactional subgraphs, the maximum throughput remains almost constant, due to the capability of FlowDB to process transactions entirely in parallel.



With more than 8 **partitions**, the throughput starts increasing linearly with the number of partitions, reaching almost 10k elements/s with 64 partitions.

Keyspace: even in the extreme case of only 10 keys, FlowDB processes close to 500 elements/s.

Queries: the keyspace size is 50, updates rate is 4500 tr/s, queries lock 10% (5) of the keyspace conflicting with concurrent updates as well as with other queries. We observe an increase in the latency after 100 queries/s.

Conclusions & Future Work

- We proposed and evaluated a new system that **integrates stream processing and data management systems**
- **Promising** performance **results**
- In the future, we will test FlowDB against **real-world workloads**
- We will investigate **fault tolerance** and the possibility to specify **different levels of isolation**
- **Optimistic protocols**

Q&A