Accelerating Confidential Databases with Crypto-free Mappings

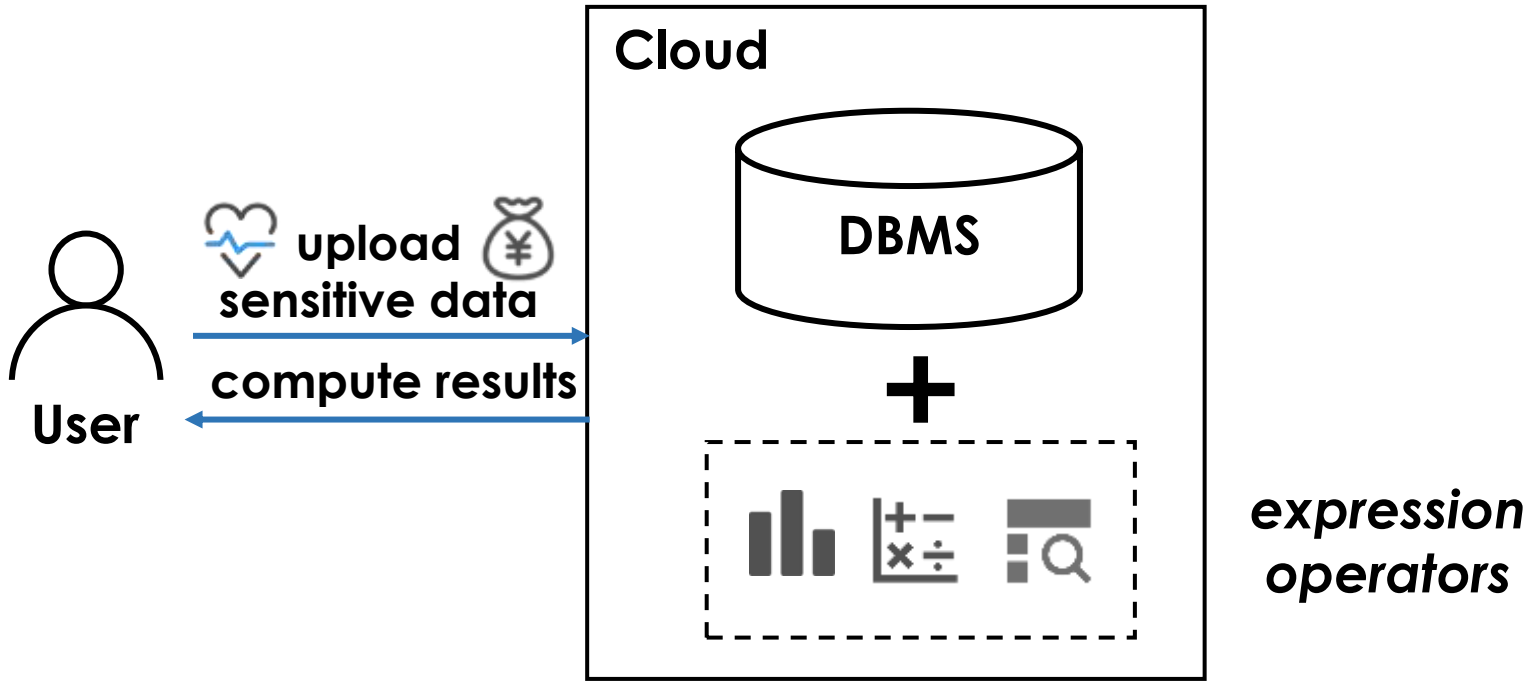
Wenxuan Huang, Zhanbo Wang, Mingyu Li

Key Laboratory of System Software (Chinese Academy of Sciences)
Institute of Software, Chinese Academy of Sciences
University of Chinese Academy of Sciences



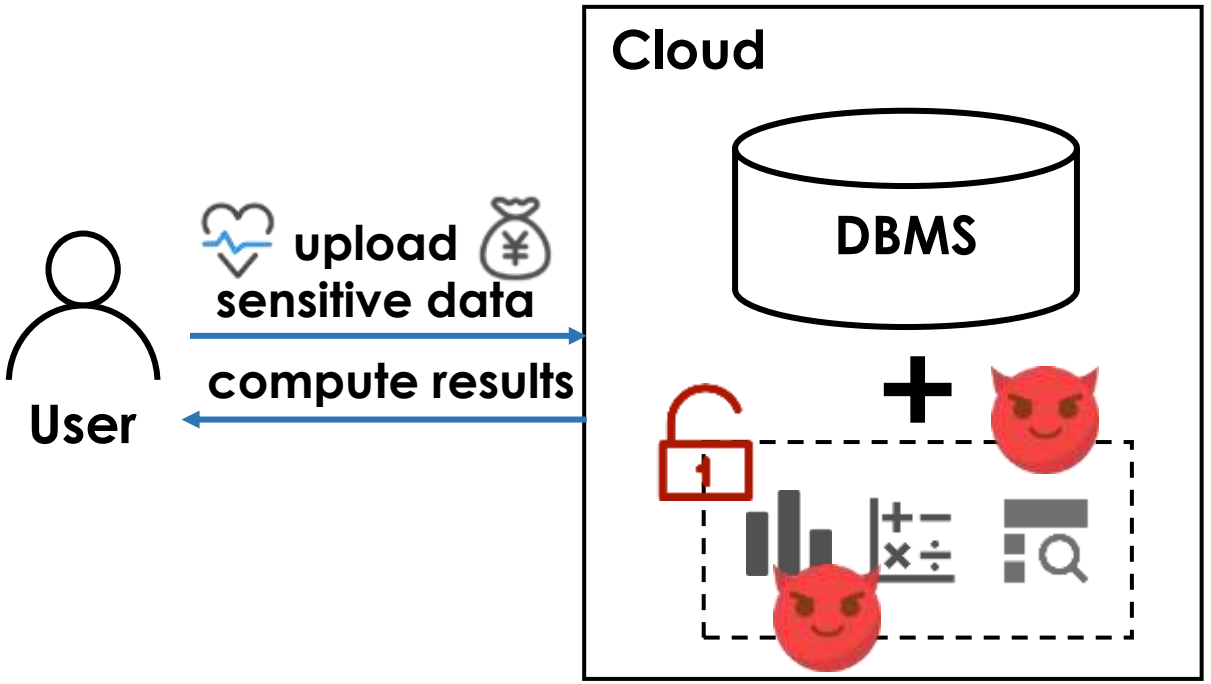
Confidential databases (CDBs)

- DBaaS (Database-as-a-Service)



Confidential databases (CDBs)

- DBaaS (Database-as-a-Service) runs on untrusted infrastructure.

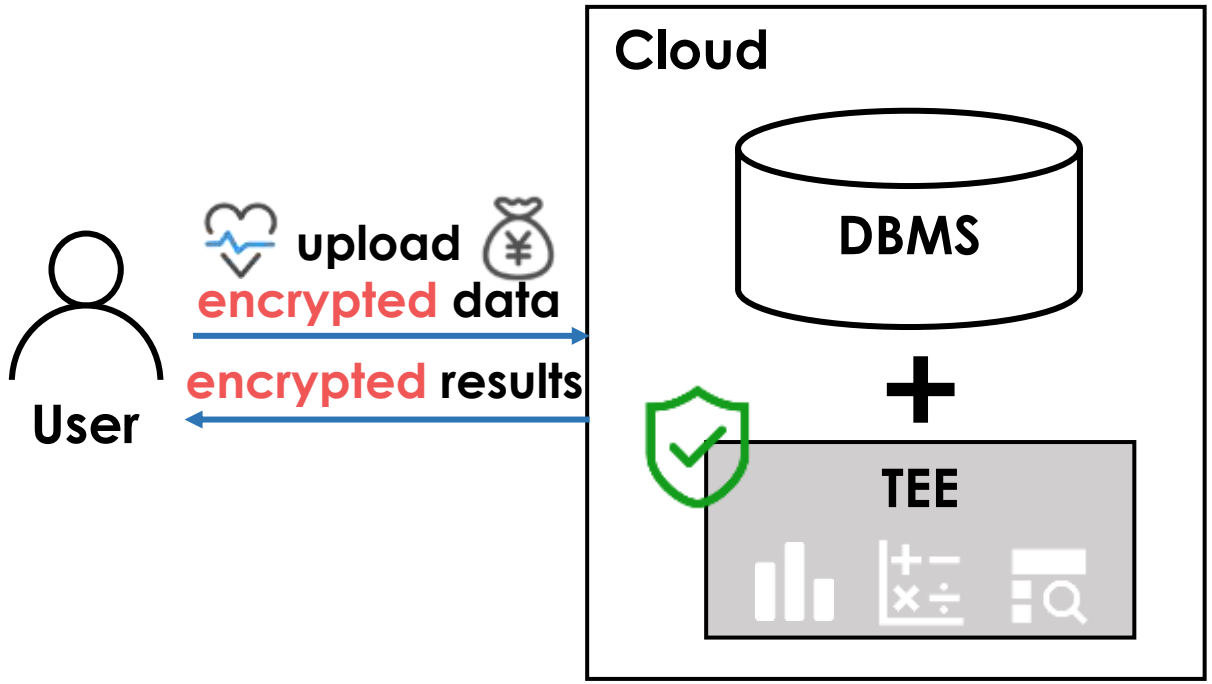


e.g., malicious admins, compromised hypervisors

data breach risk

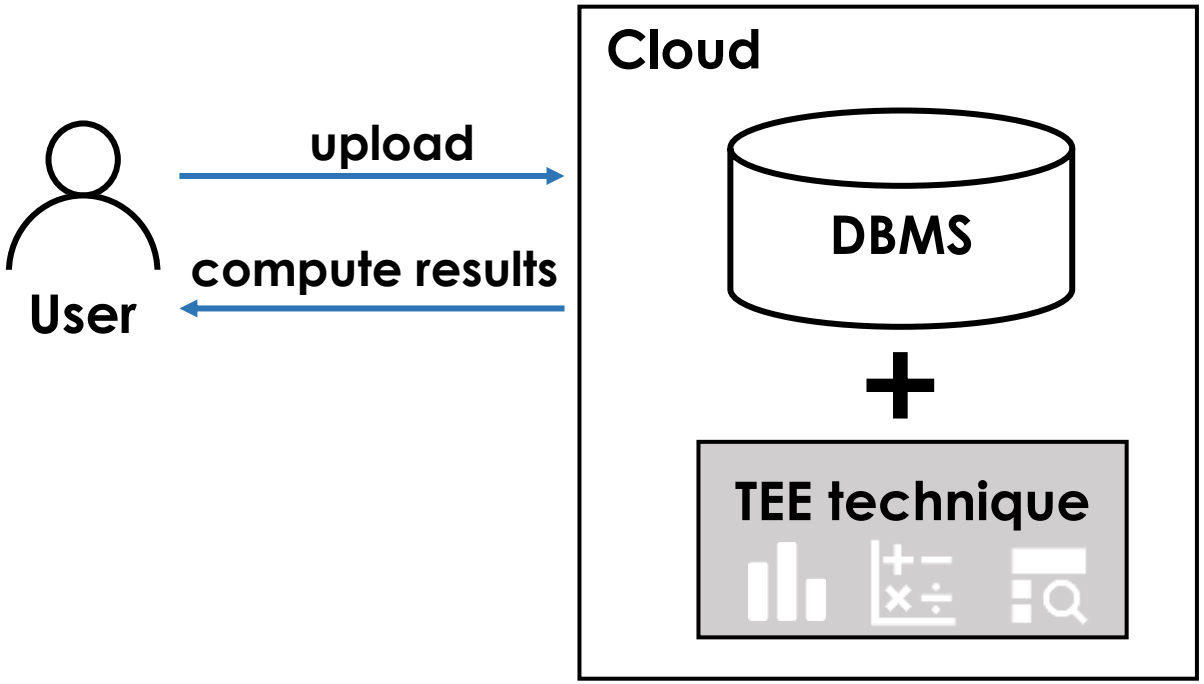
Confidential databases (CDBs)

- DBaaS (Database-as-a-Service) runs on untrusted infrastructure.
- CDBs leverage Trusted Execution Environment (**TEE**) hardware to isolate sensitive data and computation.



any sensitive data leaving TEEs must be encrypted

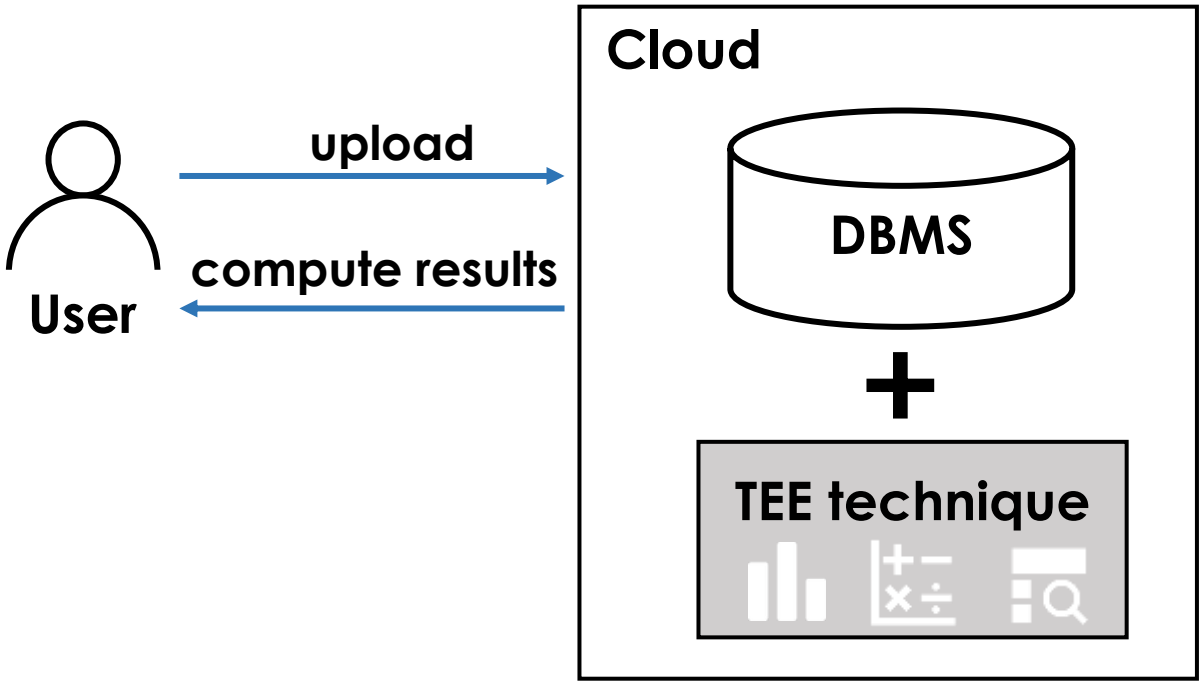
Design goals of CDBs



Design goals of CDBs



Efficiency: Narrow the performance gap relative to plaintext databases.

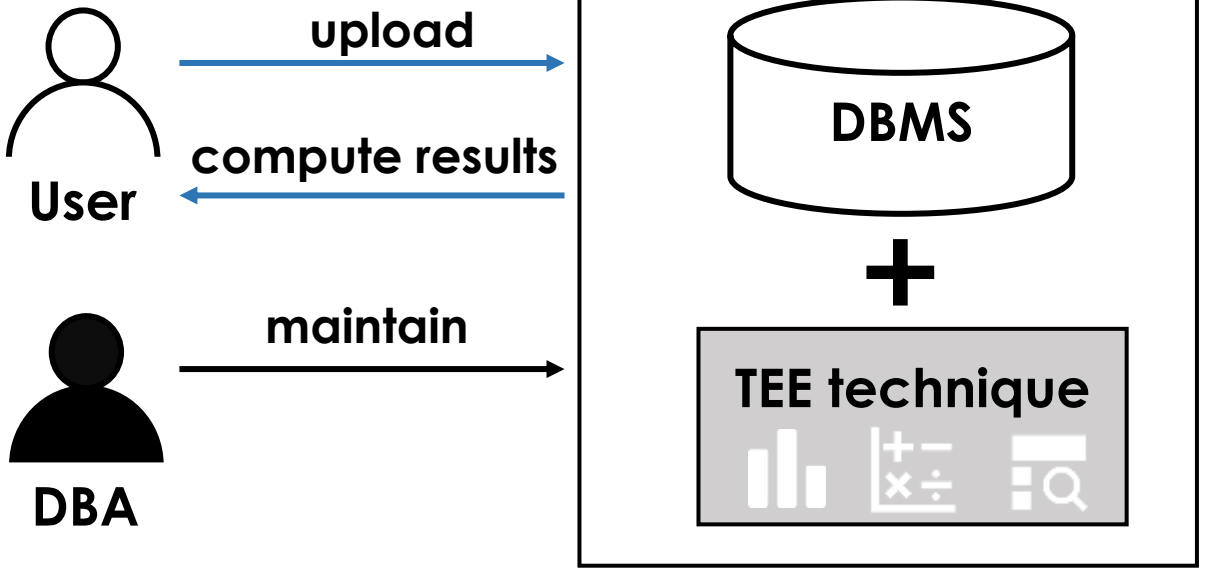


Design goals of CDBs



Efficiency: Narrow the performance gap relative to plaintext databases.

Maintenance: Allow database administrators (DBAs) to diagnose DBMS state.

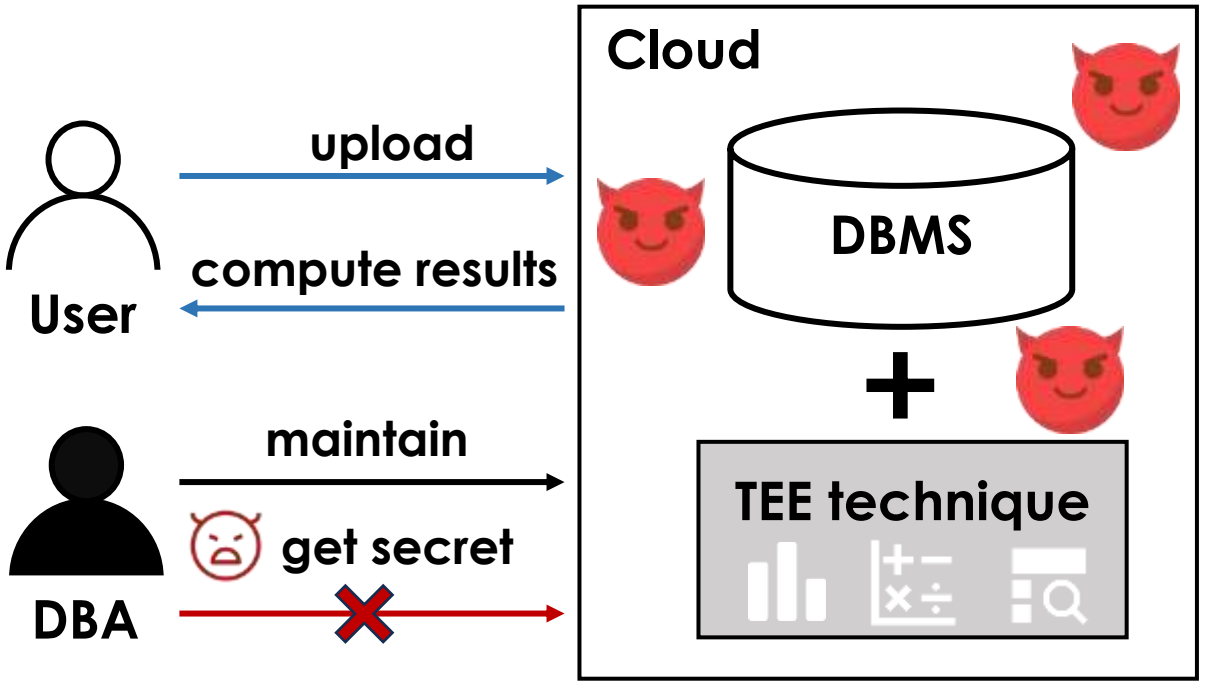


Design goals of CDBs



Efficiency: Narrow the performance gap relative to plaintext databases.

Maintenance: Allow database administrators (DBAs) to diagnose DBMS state.



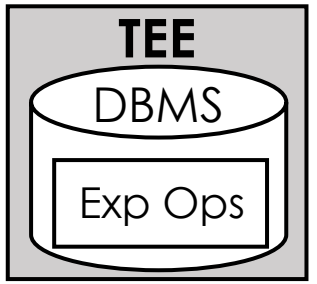
Security: Protect data confidentiality from compromised infrastructure and curious DBAs.

Existing CDB designs

	Security	Maintenance	Efficiency
(a) Monolithic CDBs **TrustedDB**@SIGMOD11, **EnclaveDB**@S&P18 **DBStore**@ICETE18, **SecuDB**@VLDB24			
(b) Split-Arch CDBs **GaussDB**@VLDB21, **Operon**@VLDB22 **AlwaysEncrypted**@SIGMOD20			
└— **HEDB** **HEDB**@OSDI23			

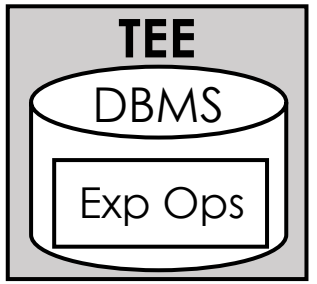
Existing CDB designs

	Security	Maintenance	Efficiency
(a) Monolithic CDBs **TrustedDB**@SIGMOD11, **EnclaveDB**@S&P18 **DBStore**@ICETE18, **SecuDB**@VLDB24	✓	✗	✓
(b) Split-Arch CDBs **GaussDB**@VLDB21, **Operon**@VLDB22 **AlwaysEncrypted**@SIGMOD20			
└— **HEDB** **HEDB**@OSDI23			



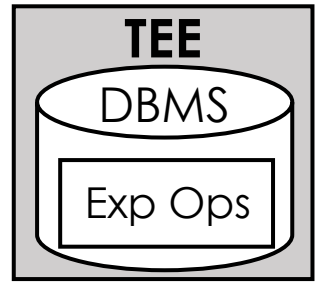
Existing CDB designs

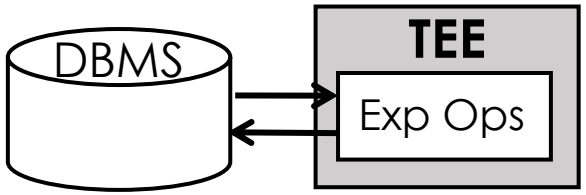
	Security	Maintenance	Efficiency
(a) Monolithic CDBs **TrustedDB**@SIGMOD11, **EnclaveDB**@S&P18 **DBStore**@ICETE18, **SecuDB**@VLDB24	✓	✗ TEE black box	✓
(b) Split-Arch CDBs **GaussDB**@VLDB21, **Operon**@VLDB22 **AlwaysEncrypted**@SIGMOD20			
└— **HEDB** **HEDB**@OSDI23			



Existing CDB designs

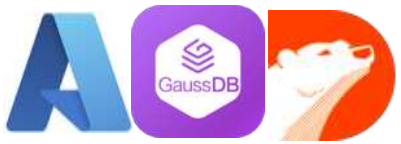
	Security	Maintenance	Efficiency
(a) Monolithic CDBs **TrustedDB**@SIGMOD11, **EnclaveDB**@S&P18 **DBStore**@ICETE18, **SecuDB**@VLDB24	✓	✗ (TEE black box)	✓
(b) Split-Arch CDBs **GaussDB**@VLDB21, **Operon**@VLDB22 **AlwaysEncrypted**@SIGMOD20	✗	✓	✗





└ **HEDB**

HEDB@OSDI23

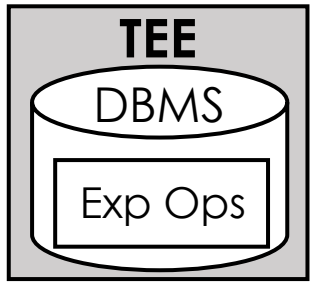


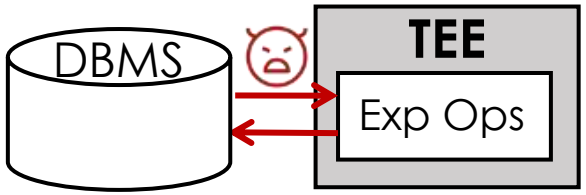
Modern CDBs

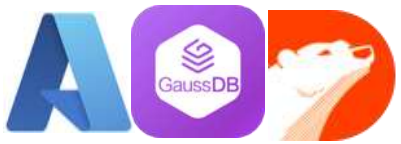
OSLAB ChSys ios.ac.cn

Existing CDB designs

	Security	Maintenance	Efficiency
(a) Monolithic CDBs **TrustedDB**@SIGMOD11, **EnclaveDB**@S&P18 **DBStore**@ICETE18, **SecuDB**@VLDB24	✓	✗ TEE black box	✓
(b) Split-Arch CDBs **GaussDB**@VLDB21, **Operon**@VLDB22 **AlwaysEncrypted**@SIGMOD20	✗ interface insecurity	✓	✗
HEDB **HEDB**@OSDI23			





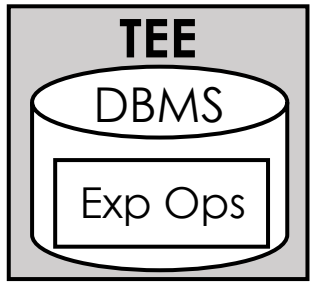


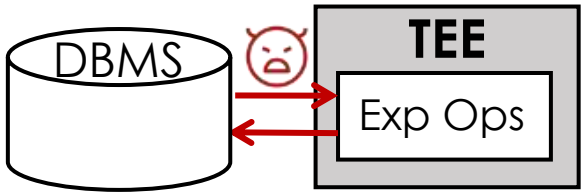
Modern CDBs

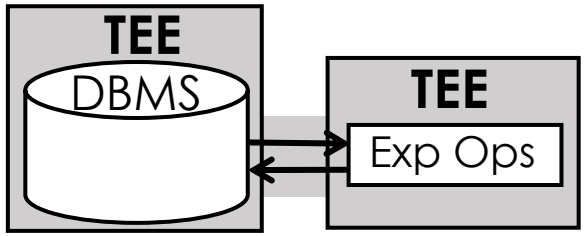
OSLAB ChSys .ios.ac.cn

Existing CDB designs

	Security	Maintenance	Efficiency
(a) Monolithic CDBs **TrustedDB**@SIGMOD11, **EnclaveDB**@S&P18 **DBStore**@ICETE18, **SecuDB**@VLDB24	✓	✗ TEE black box	✓
(b) Split-Arch CDBs **GaussDB**@VLDB21, **Operon**@VLDB22 **AlwaysEncrypted**@SIGMOD20	✗ interface insecurity	✓	✗
HEDB **HEDB**@OSDI23	✓	✓	✗



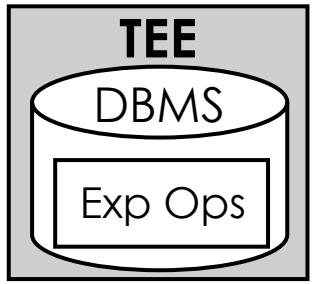


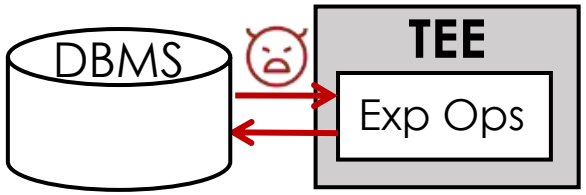


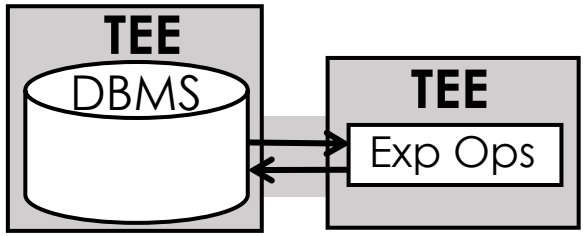
Modern CDBs

Existing CDB designs

	Security	Maintenance	Efficiency
(a) Monolithic CDBs **TrustedDB**@SIGMOD11, **EnclaveDB**@S&P18 **DBStore**@ICETE18, **SecuDB**@VLDB24	✓	✗ TEE black box	✓
(b) Split-Arch CDBs **GaussDB**@VLDB21, **Operon**@VLDB22 **AlwaysEncrypted**@SIGMOD20	✗ interface insecurity	✓	✗
HEDB **HEDB**@OSDI23	✓	✓	✗





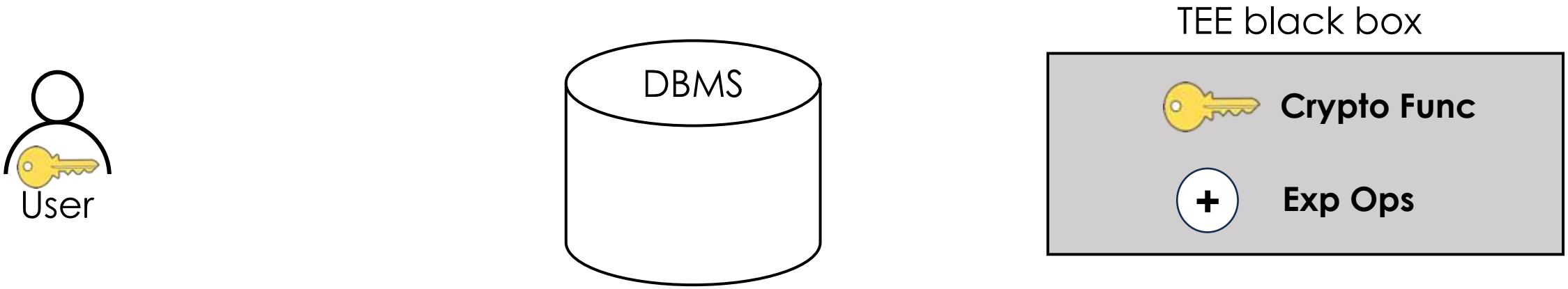


Up to 79.5× higher TPC-H latency

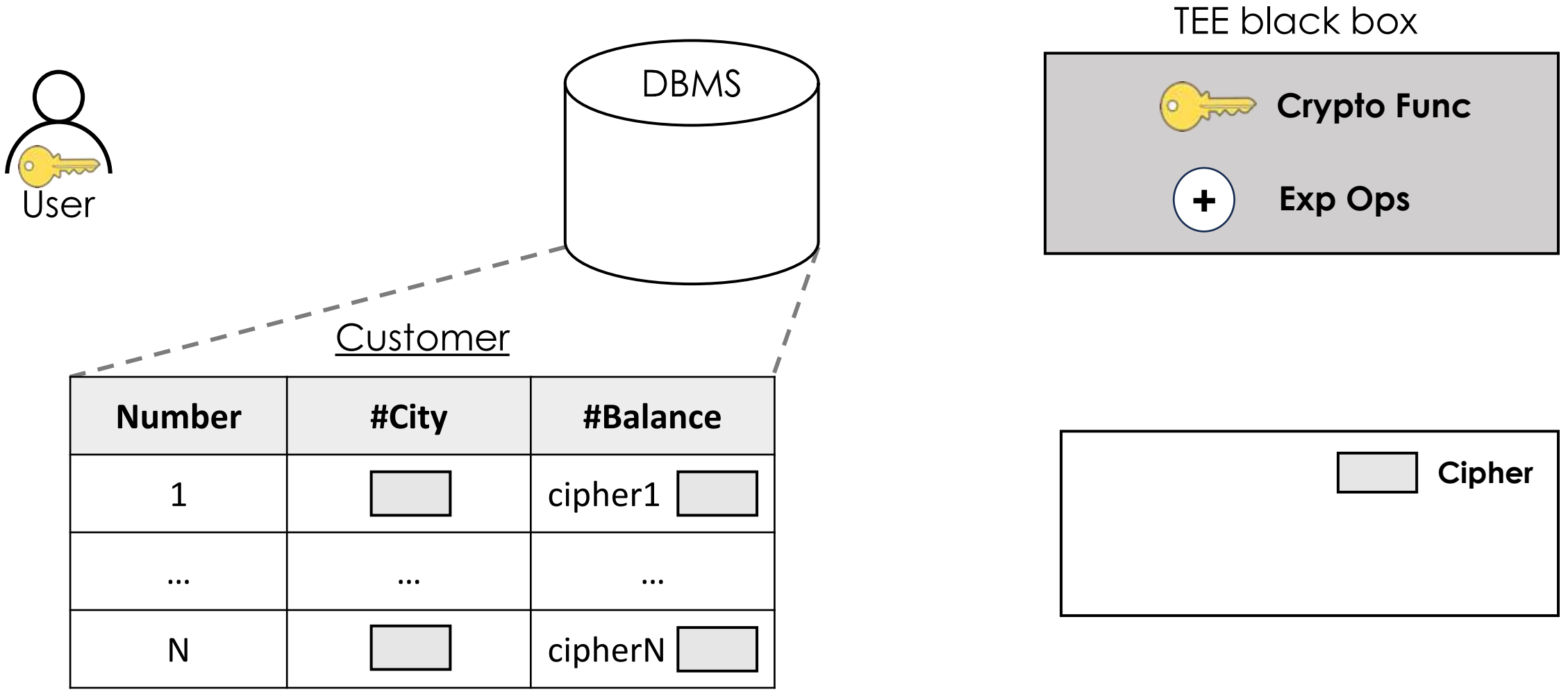
Modern CDBs

What makes modern CDBs slow?

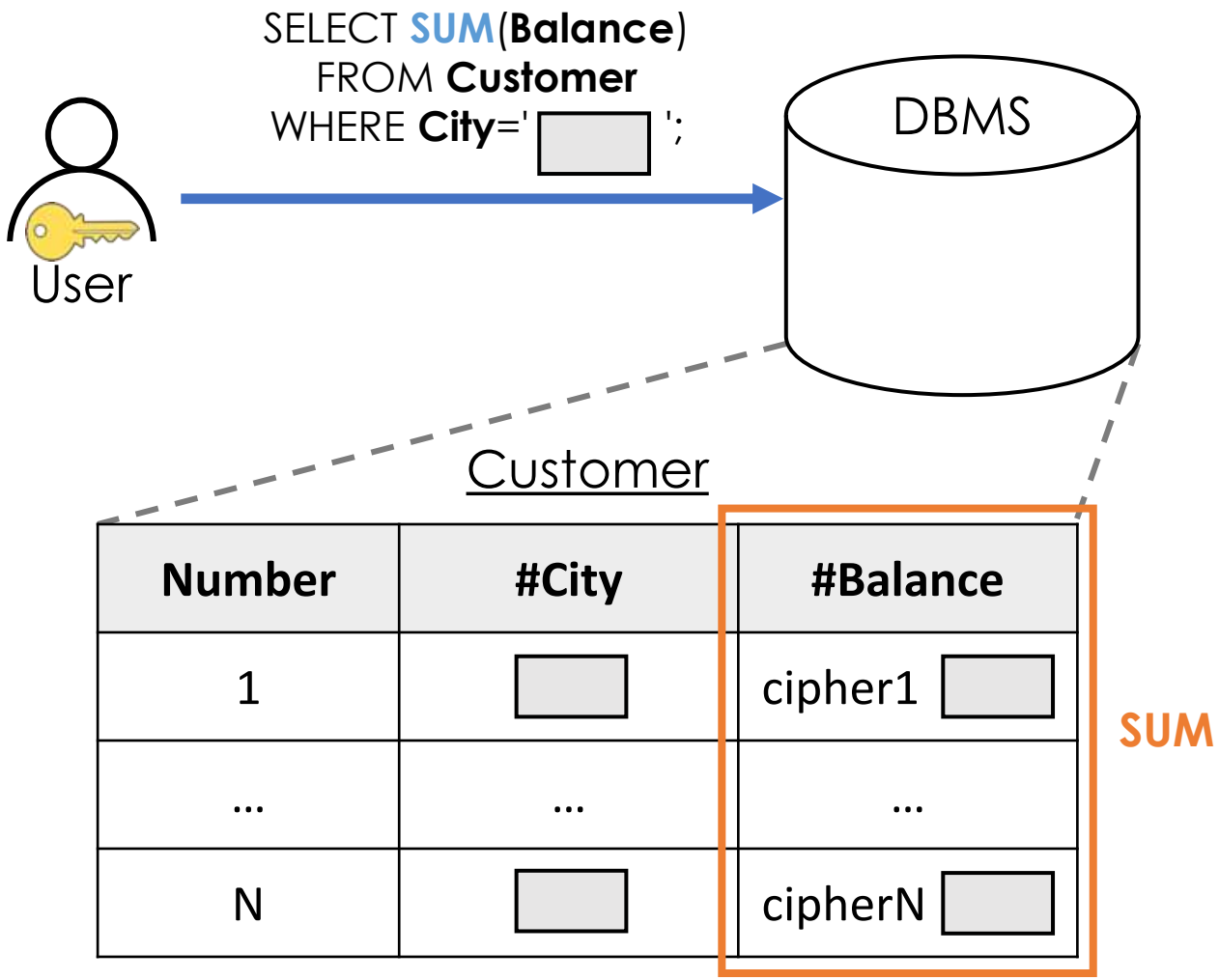
Modern CDB query workflow

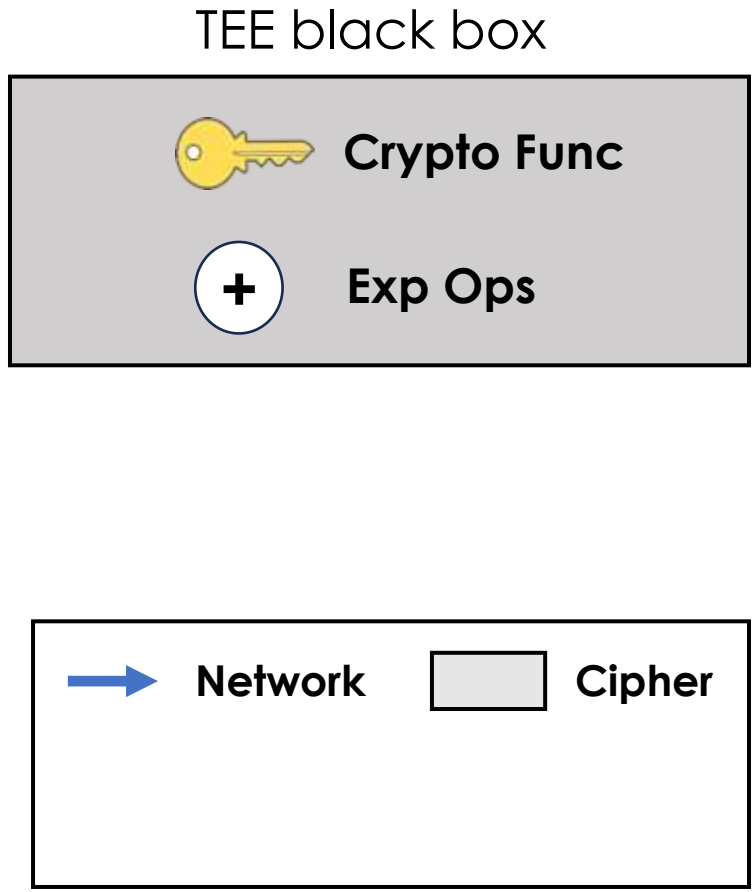


Modern CDB query workflow

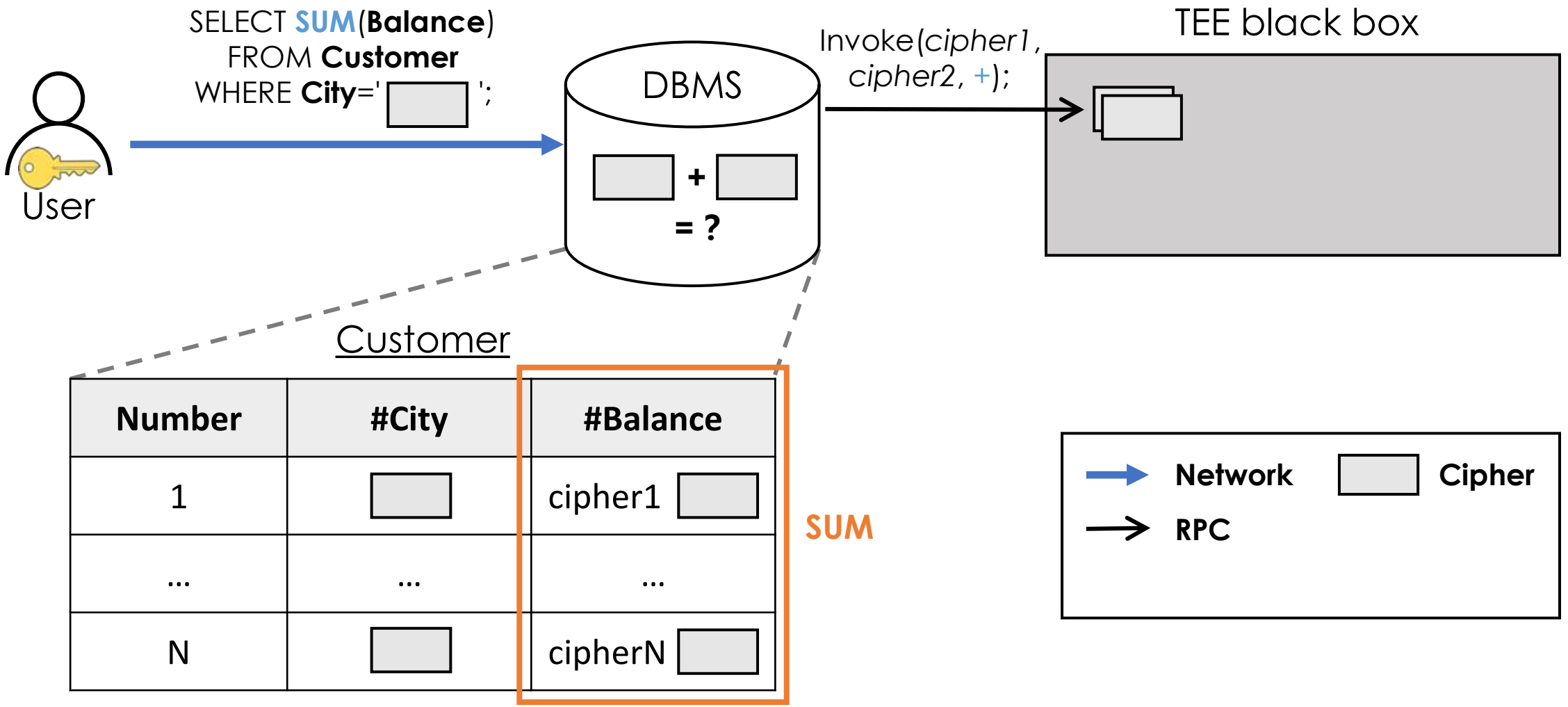


Modern CDB query workflow

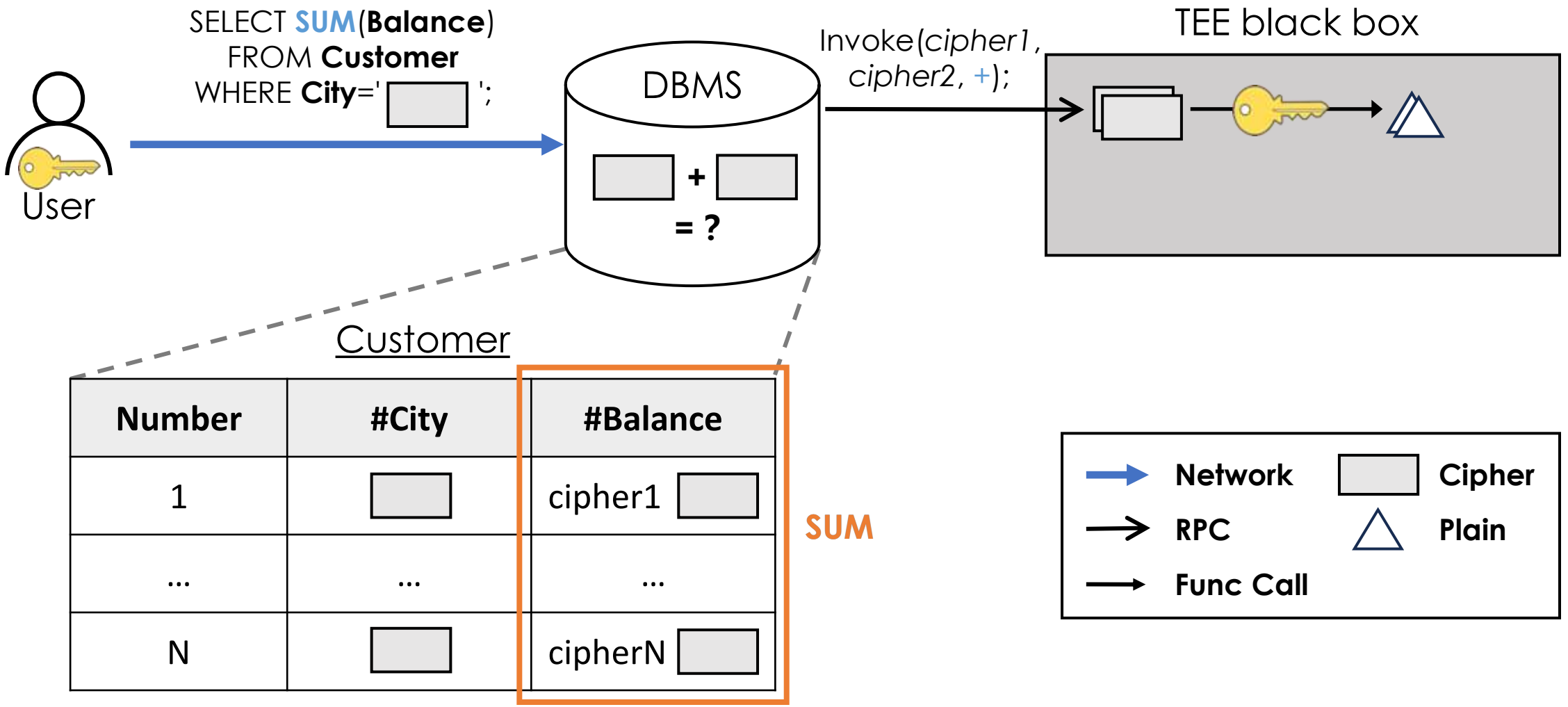




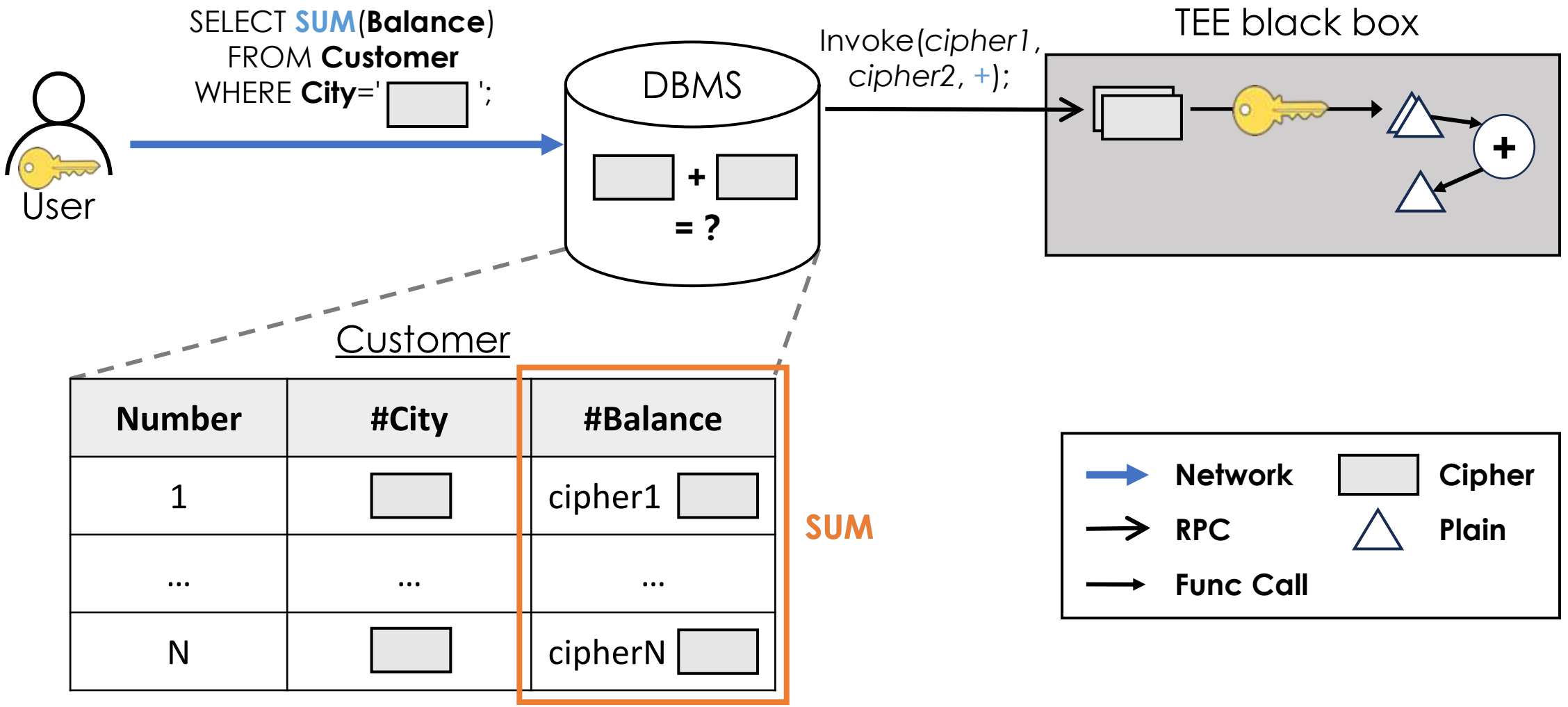
Modern CDB query workflow



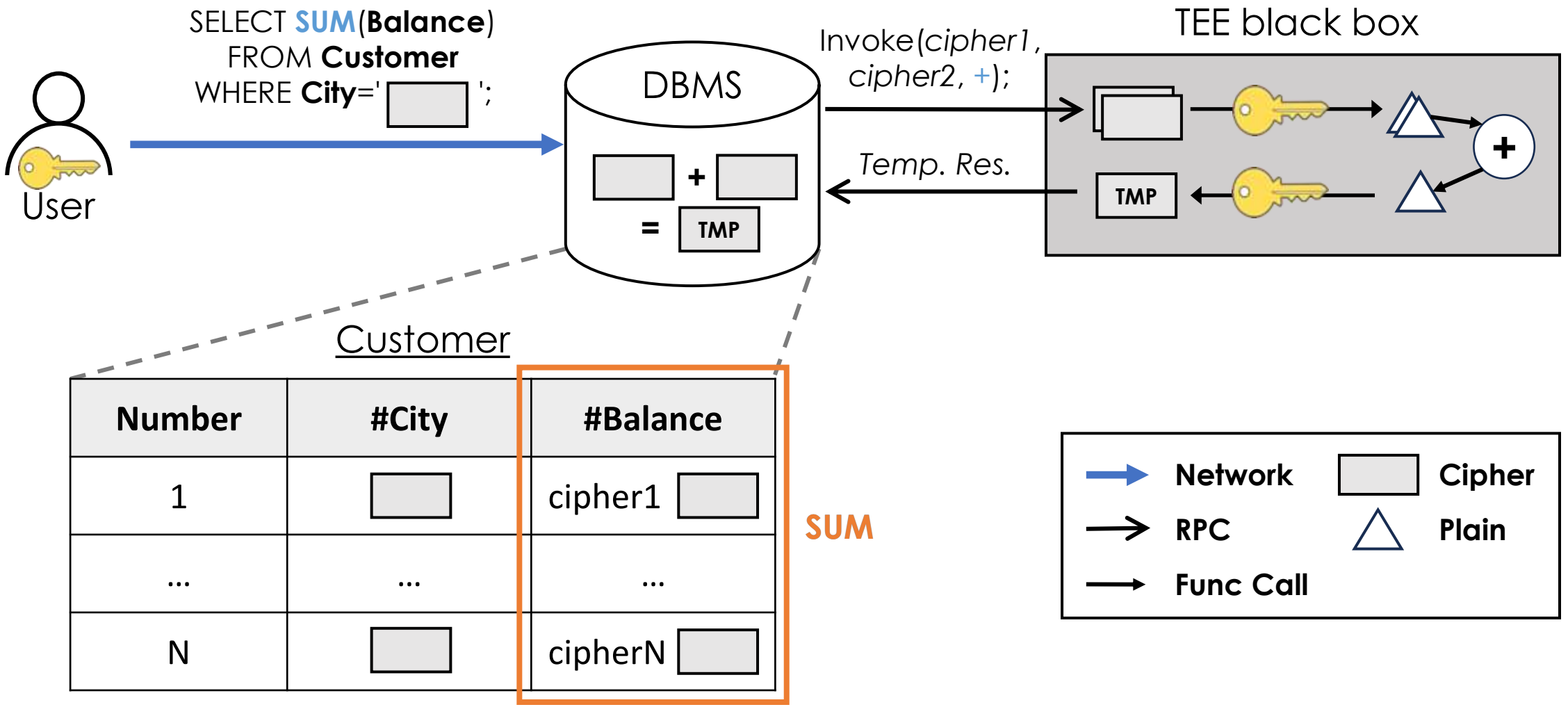
Modern CDB query workflow



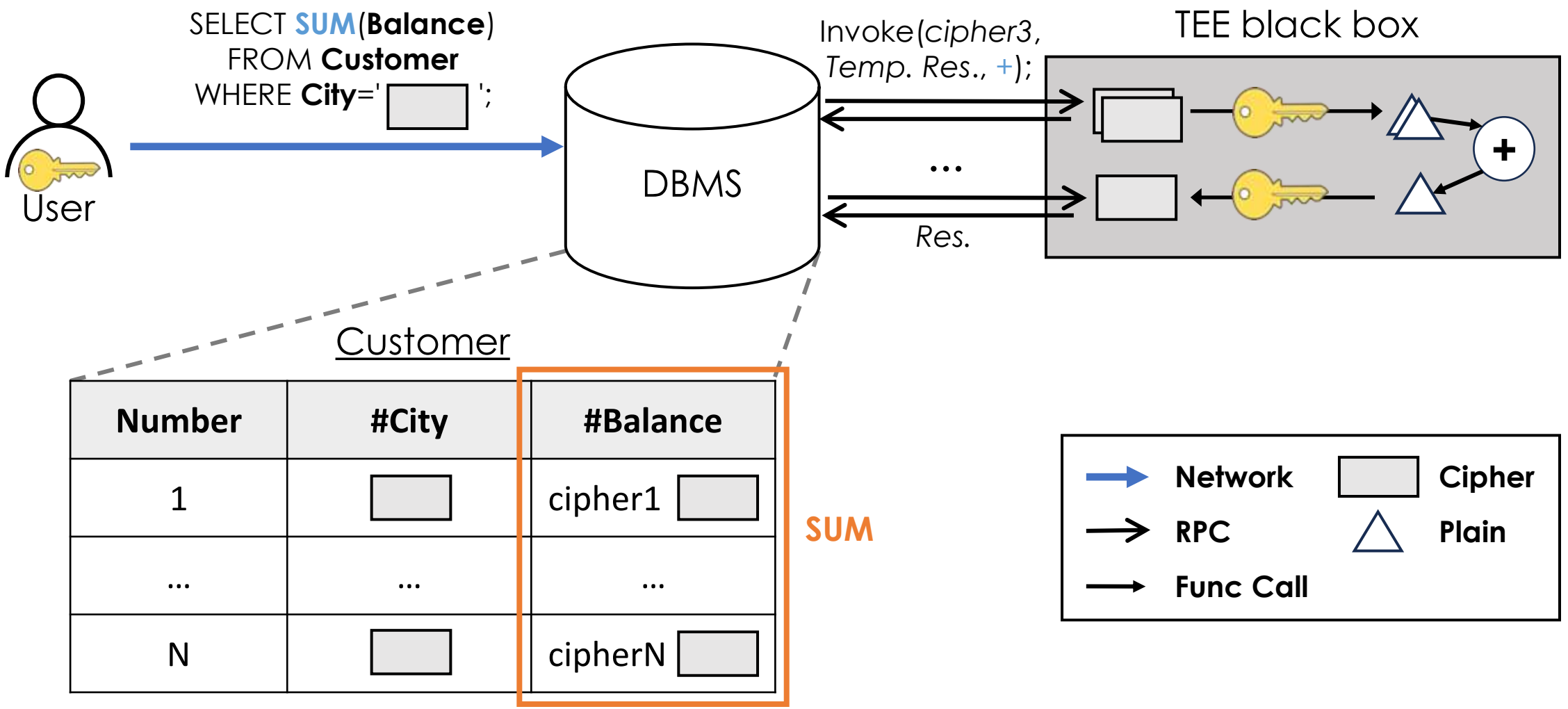
Modern CDB query workflow



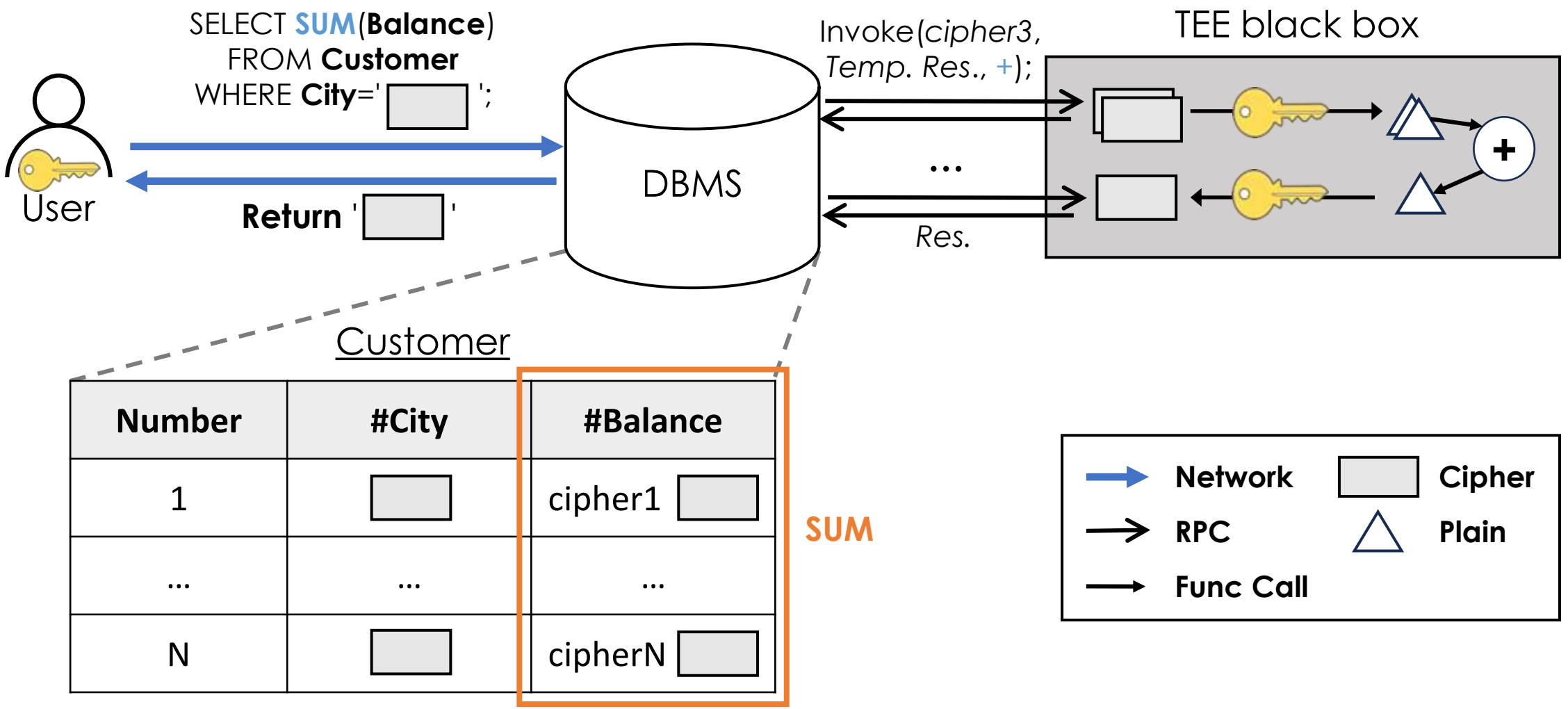
Modern CDB query workflow



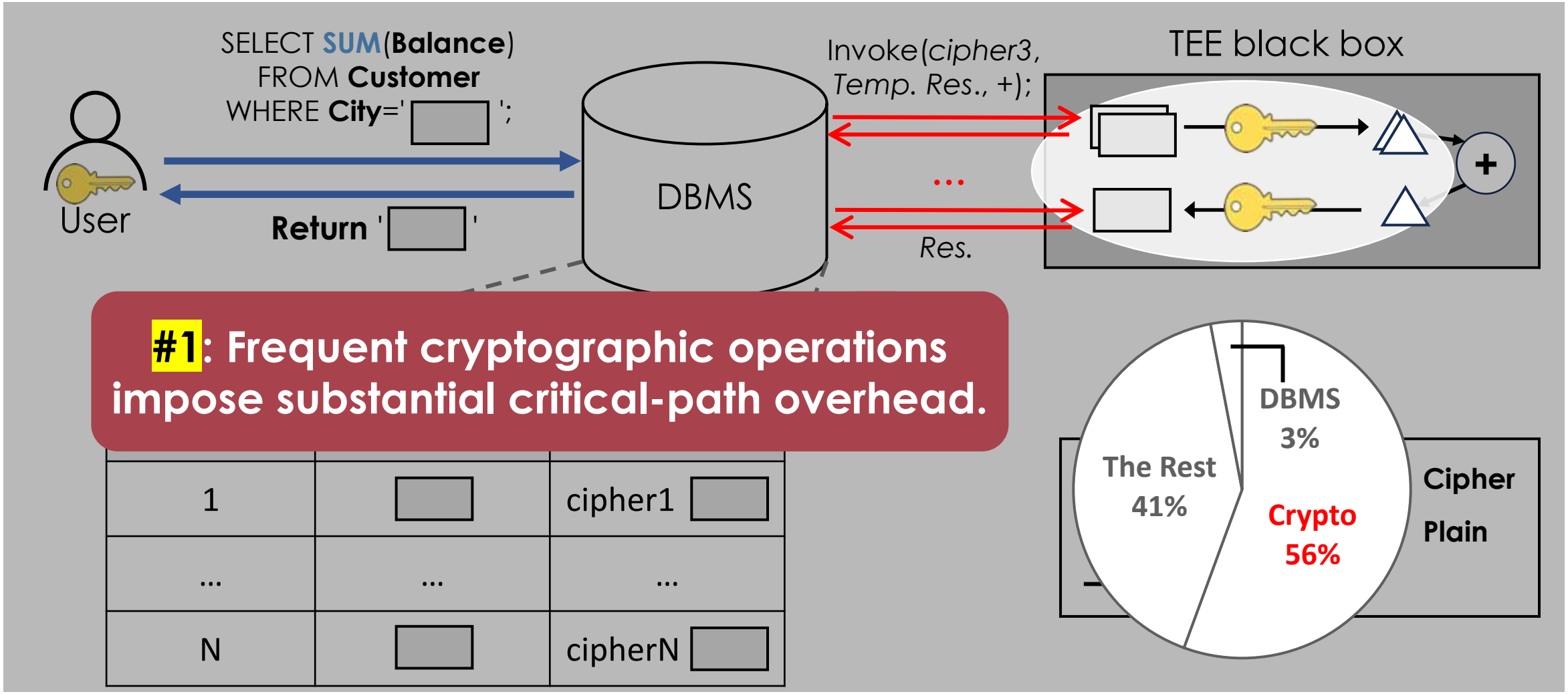
Modern CDB query workflow



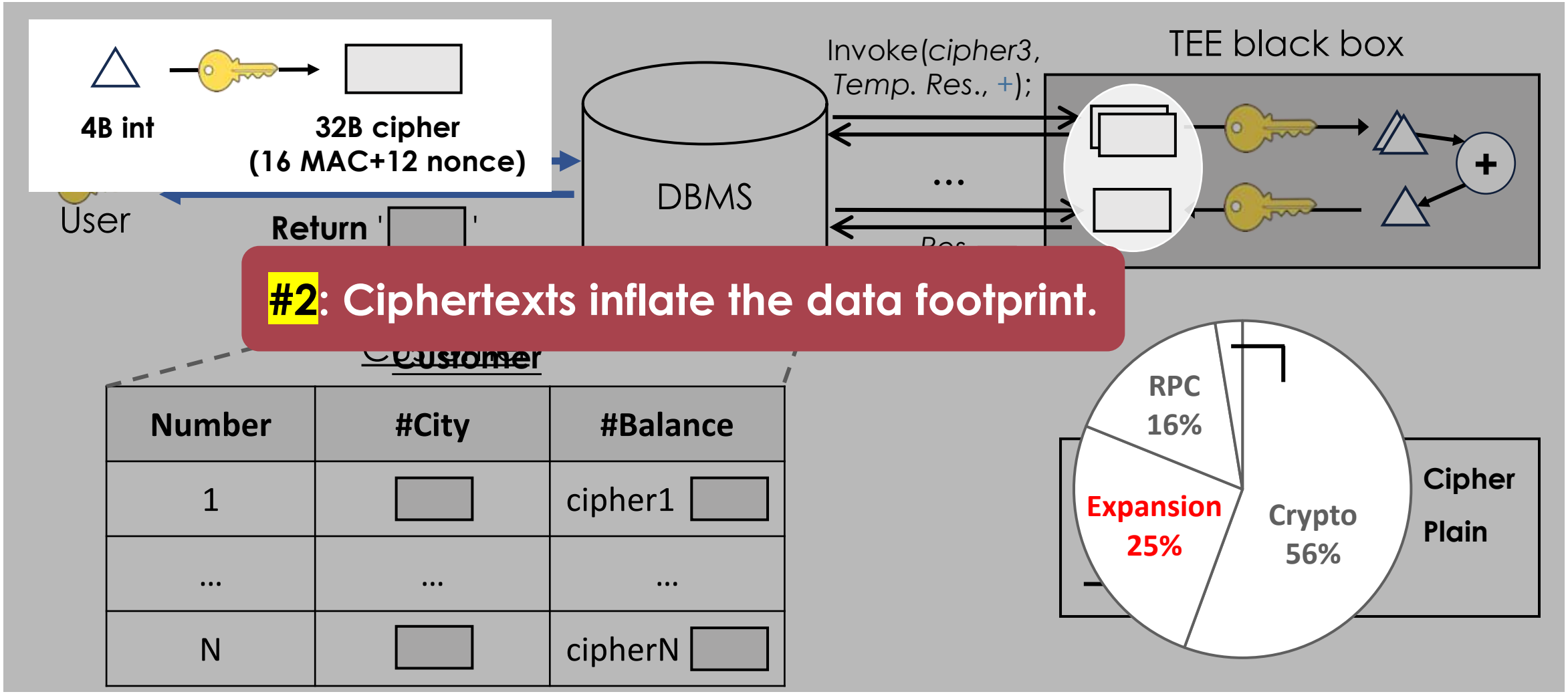
Modern CDB query workflow



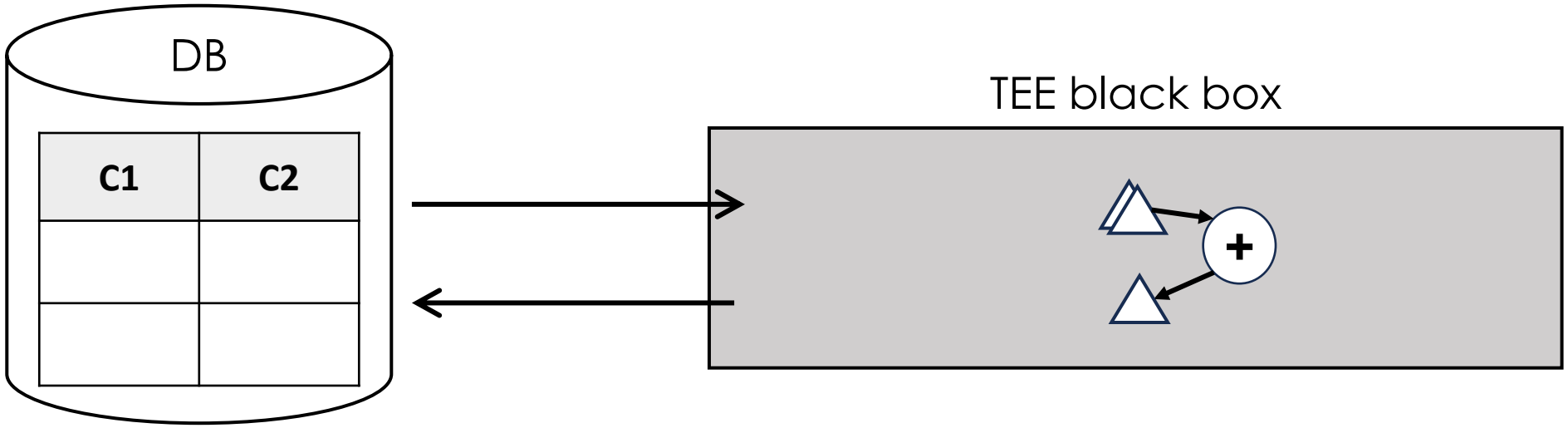
The price of ciphertexts



The price of ciphertexts

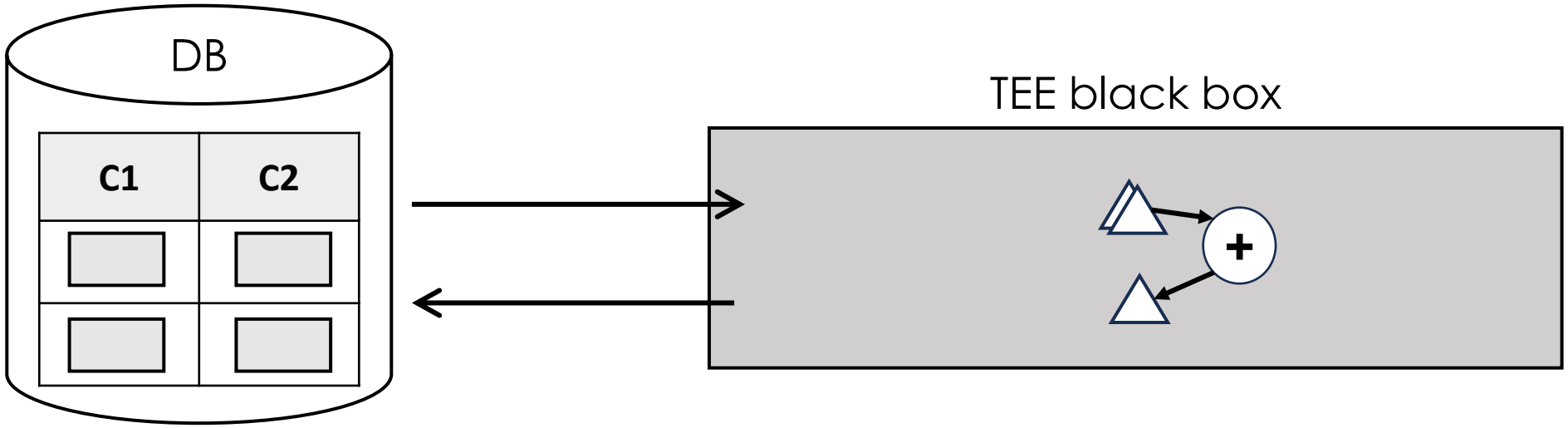


Root cause: ciphertext with indirection & protection



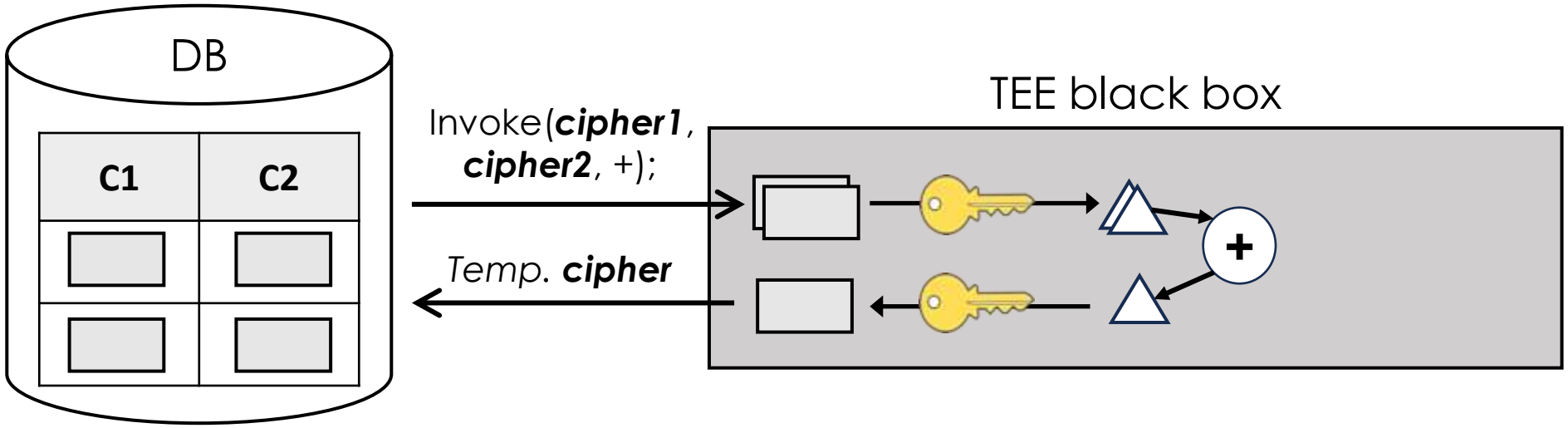
	Modern CDBs	ZENO
#1 Indirection		
#2 Protection		

Root cause: ciphertext with indirection & protection



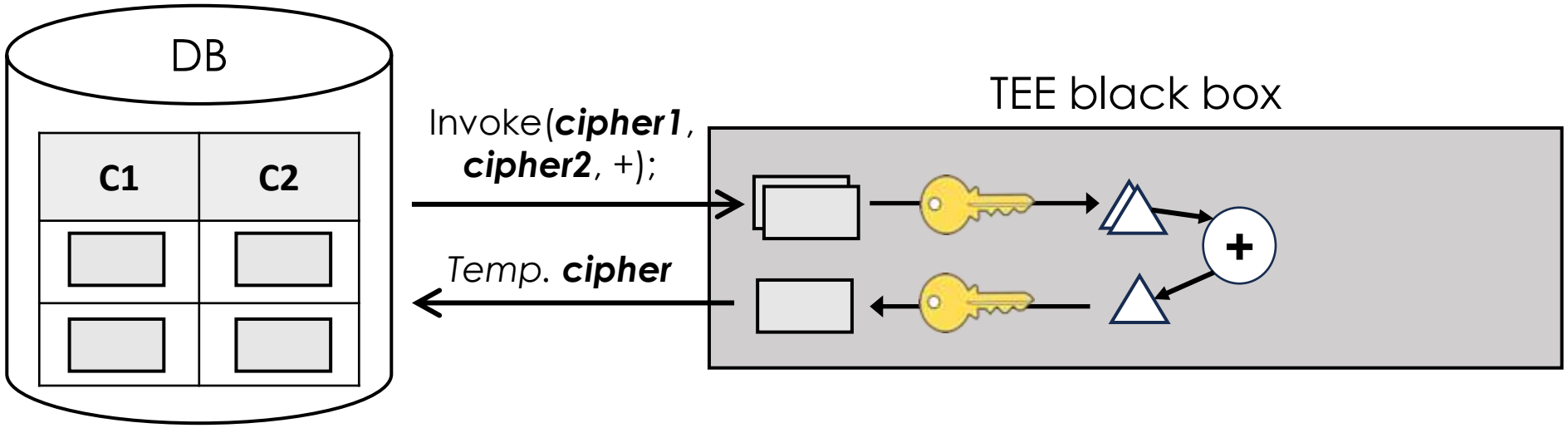
	Modern CDBs	ZENO
#1 Indirection		
#2 Protection	Ciphertexts  Any data leaving TEEs must be **encrypted**	

Root cause: ciphertext with indirection & protection



	Modern CDBs	ZENO
#1 Indirection	Ciphertexts 	
	Dec (~1,500 cycles) Enc (~5,000 cycles)	
#2 Protection	Ciphertexts 	
	Any data leaving TEEs must be **encrypted**	

Root cause: ciphertext with indirection & protection

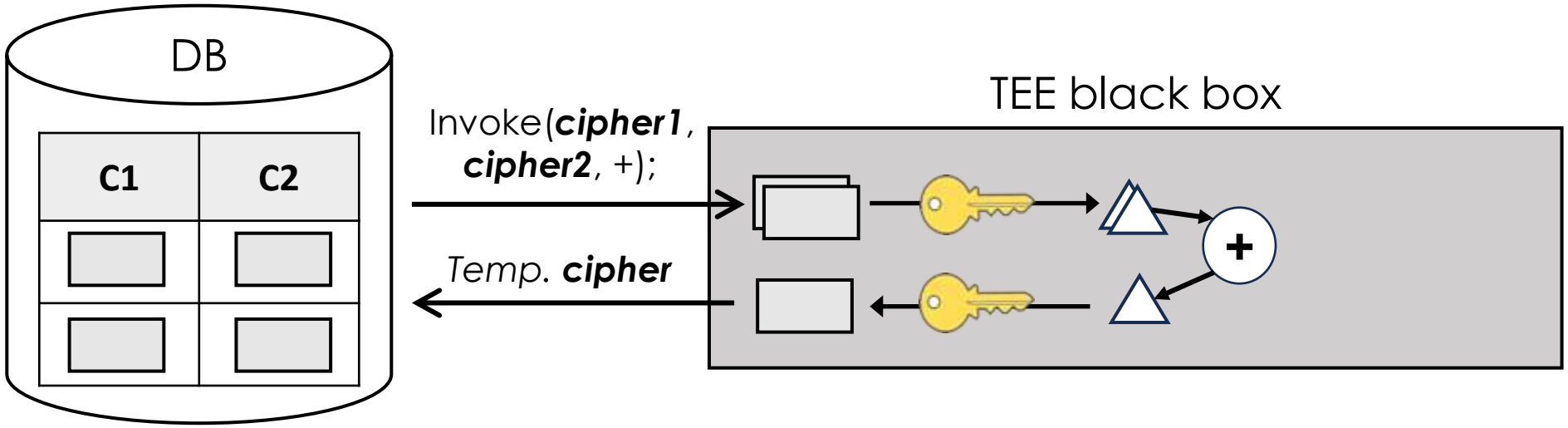


	Modern CDBs	ZENO
Coupled: #1 Indirection	Ciphertexts 	
	Dec (~1,500 cycles) Enc (~5,000 cycles)	
Coupled: #2 Protection	Ciphertexts 	
	Any data leaving TEEs must be **encrypted**	

Can we eliminate ciphertext overhead

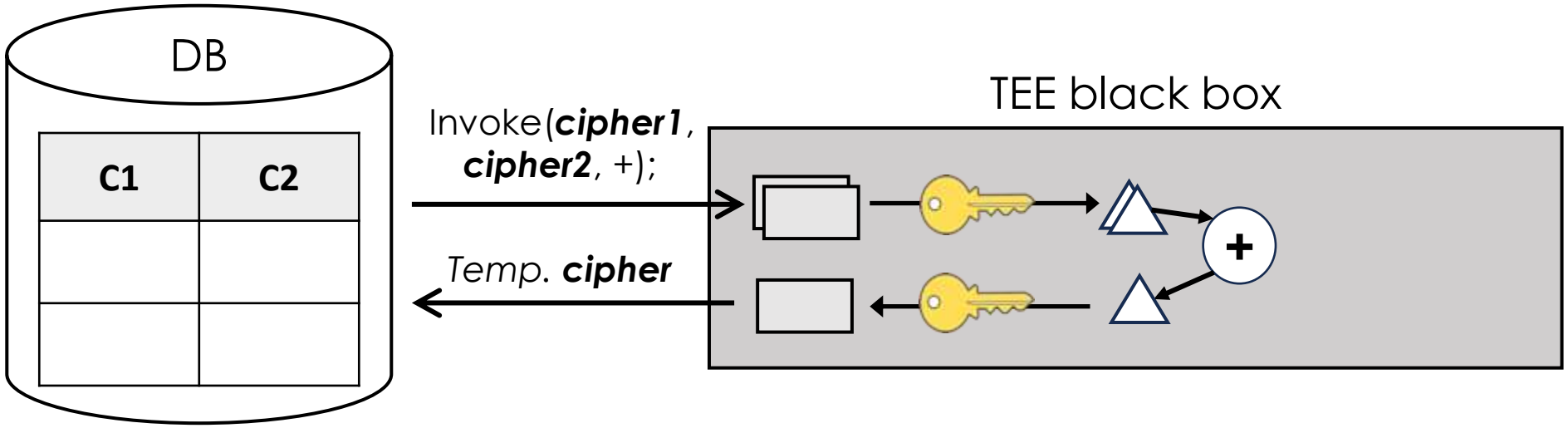
while preserving security guarantees?

Our approach: decoupling via crypto-free mappings



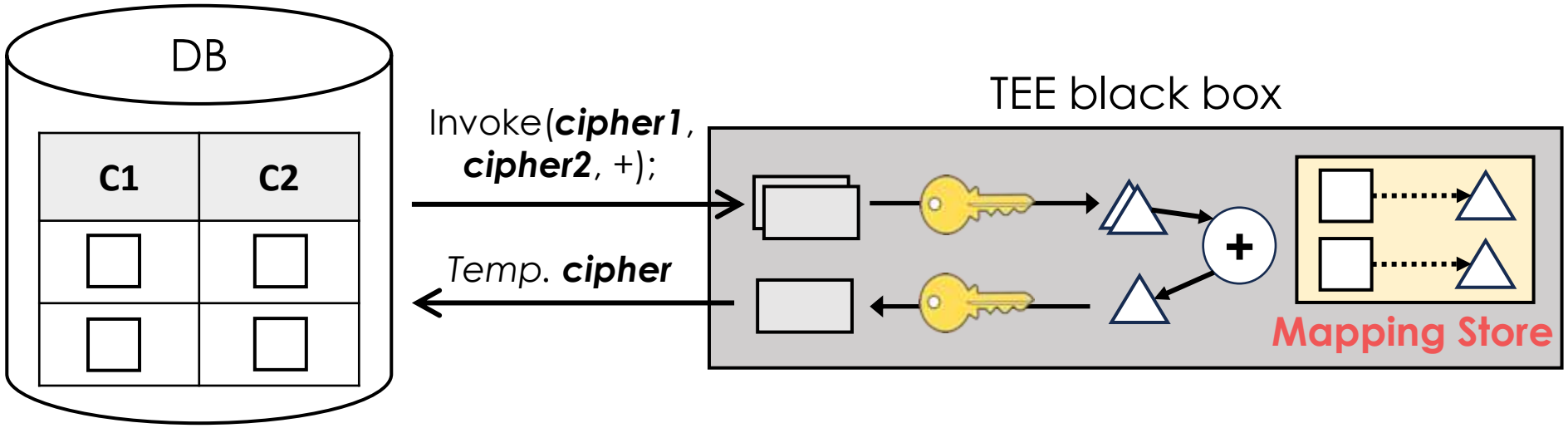
	Modern CDBs	ZENO
#1 Indirection	**Ciphertexts**  **Dec (~1,500 cycles) Enc (~5,000 cycles)**	
#2 Protection	**Ciphertexts**  Any data leaving TEEs must be **encrypted**	

Our approach: decoupling via crypto-free mappings



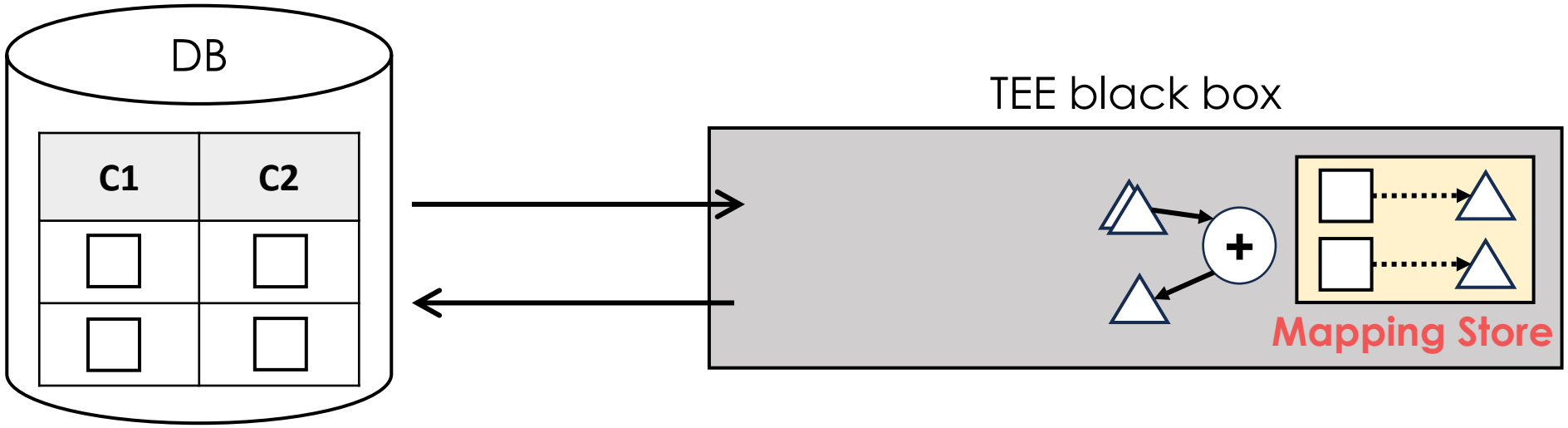
	Modern CDBs	ZENO
#1 Indirection	Ciphertexts 	
	Dec (~1,500 cycles) Enc (~5,000 cycles)	
#2 Protection	Ciphertexts 	
	Any data leaving TEEs must be **encrypted**	

Our approach: decoupling via crypto-free mappings



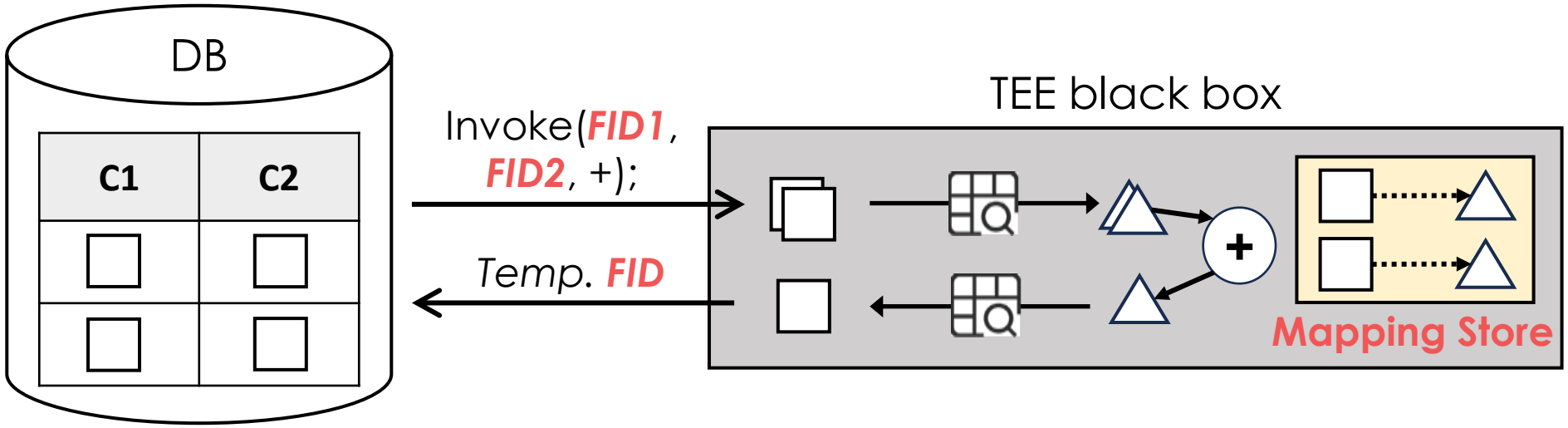
	Modern CDBs	ZENO
#1 Indirection	Ciphertexts 	
	Dec (~1,500 cycles) Enc (~5,000 cycles)	
#2 Protection	Ciphertexts 	FIDs * *Field-Identifier*
	Any data leaving TEEs must be **encrypted**	Any data leaving TEEs must be **plaintext-independent**

Our approach: decoupling via crypto-free mappings



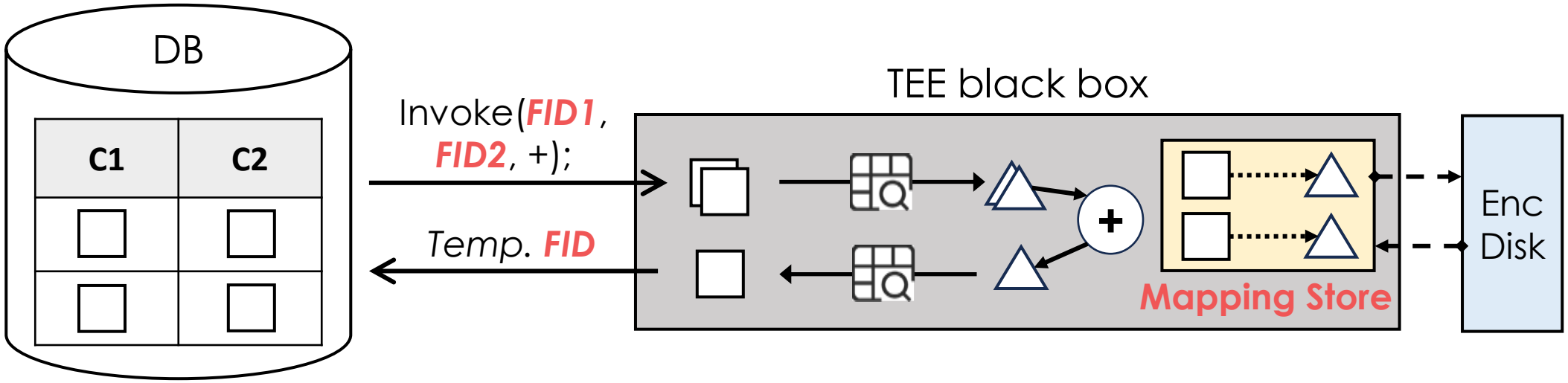
	Modern CDBs	ZENO
#1 Indirection	Ciphertexts 	
	Dec (~1,500 cycles) Enc (~5,000 cycles)	
#2 Protection	Ciphertexts 	FIDs * *Field-Identifier*
	Any data leaving TEEs must be **encrypted**	Any data leaving TEEs must be **plaintext-independent**

Our approach: decoupling via crypto-free mappings



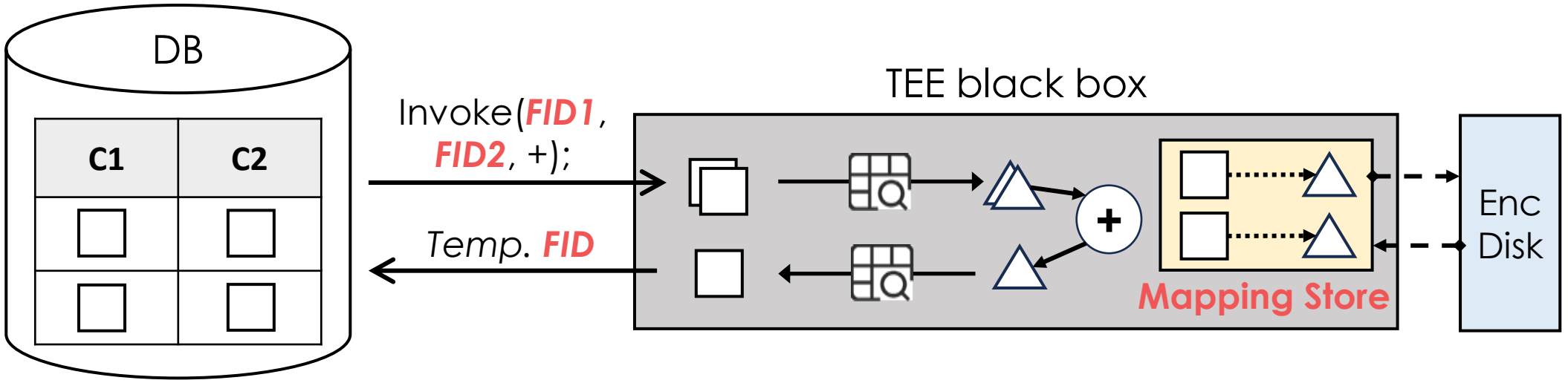
	Modern CDBs	ZENO
#1 Indirection	Ciphertexts	FIDs
	Dec (~1,500 cycles) Enc (~5,000 cycles)	Map Get (~30 cycles) Map Put (~200 cycles)
#2 Protection	Ciphertexts	FIDs * *Field-Identifier*
	Any data leaving TEEs must be **encrypted**	Any data leaving TEEs must be **plaintext-independent**

Our approach: decoupling via crypto-free mappings



	Modern CDBs	ZENO
#1 Indirection	Ciphertexts	FIDs
	Dec (~1,500 cycles) Enc (~5,000 cycles)	Map Get (~30 cycles) Map Put (~200 cycles)
#2 Protection	Ciphertexts	FIDs + batched encryption at rest
	Any data leaving TEEs must be **encrypted**	Any data leaving TEEs must be **encrypted** or plaintext-independent

Our approach: decoupling via crypto-free mappings



		Modern CDBs	ZENO
Decoupled	#1 Indirection	Ciphertexts	FIDs
		Dec (~1,500 cycles) Enc (~5,000 cycles)	Map Get (~30 cycles) Map Put (~200 cycles)
	#2 Protection	Ciphertexts	FIDs + batched encryption at rest
		Any data leaving TEEs must be **encrypted**	Any data leaving TEEs must be **encrypted** or plaintext-independent

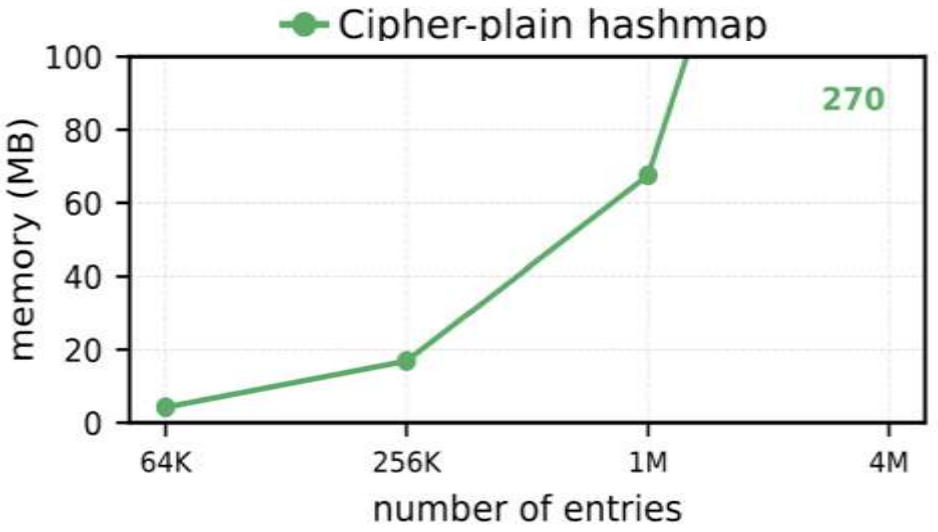
Challenges

- **#1: How to build a scalable mapping store?**
  - 🎈 Level 1: Scaling in memory
  - 🎈 Level 2: Scaling to disk

- **#2: How to ensure transactional consistency between FIDs and their mappings?**

Level 1: Scaling in memory

- FID representation considerations:
  - ❌ Ciphertext FIDs: memory bloat

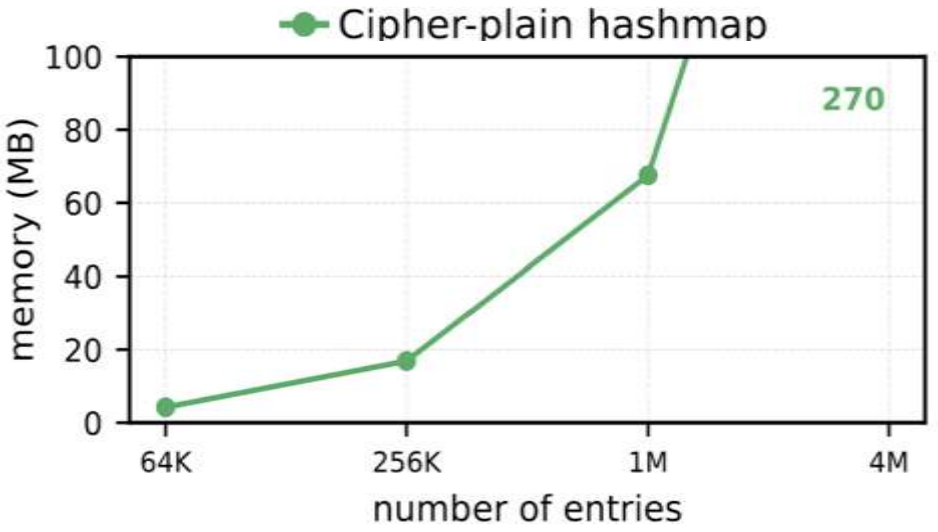


Level 1: Scaling in memory

- FID representation considerations:
  - ❌ Ciphertext FIDs: memory bloat
  - ✅ ZENO's Choice: integer-based FIDs

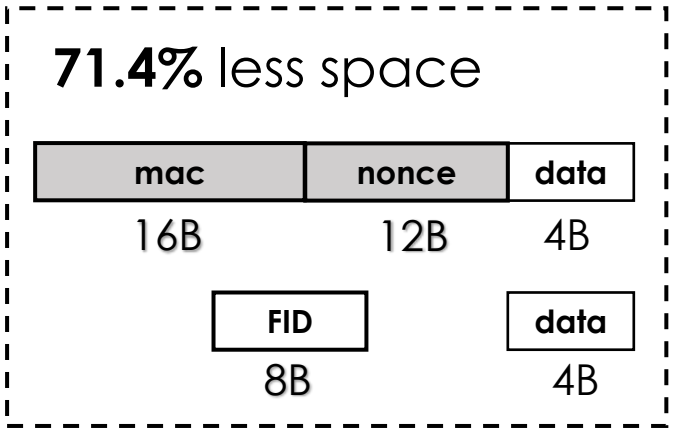
0x 0000 0000 0000 0001

Number	#Balance
1	1
...	...
N	N



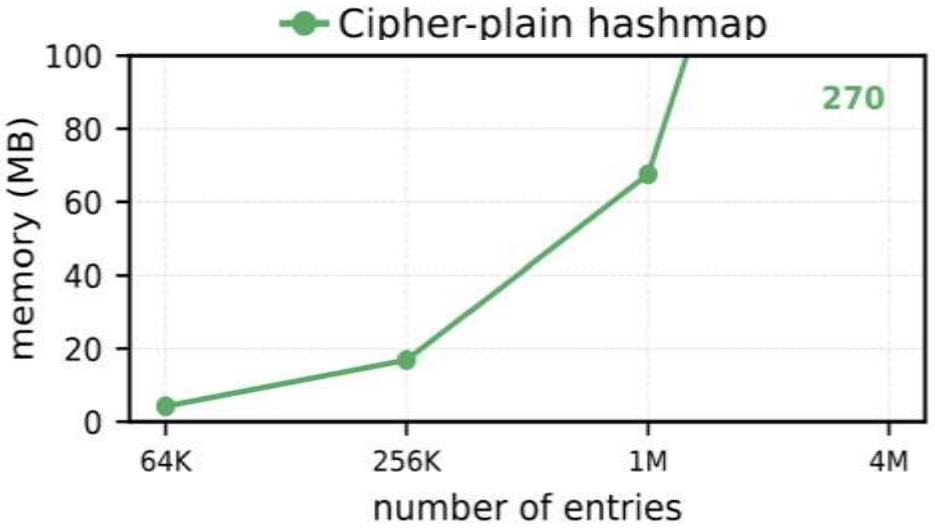
Level 1: Scaling in memory

- FID representation considerations:
  - ❌ Ciphertext FIDs: memory bloat
  - ✅ ZENO's Choice: integer-based FIDs



0x 0000 0000 0000 0001

Number	#Balance
1	1
...	...
N	N

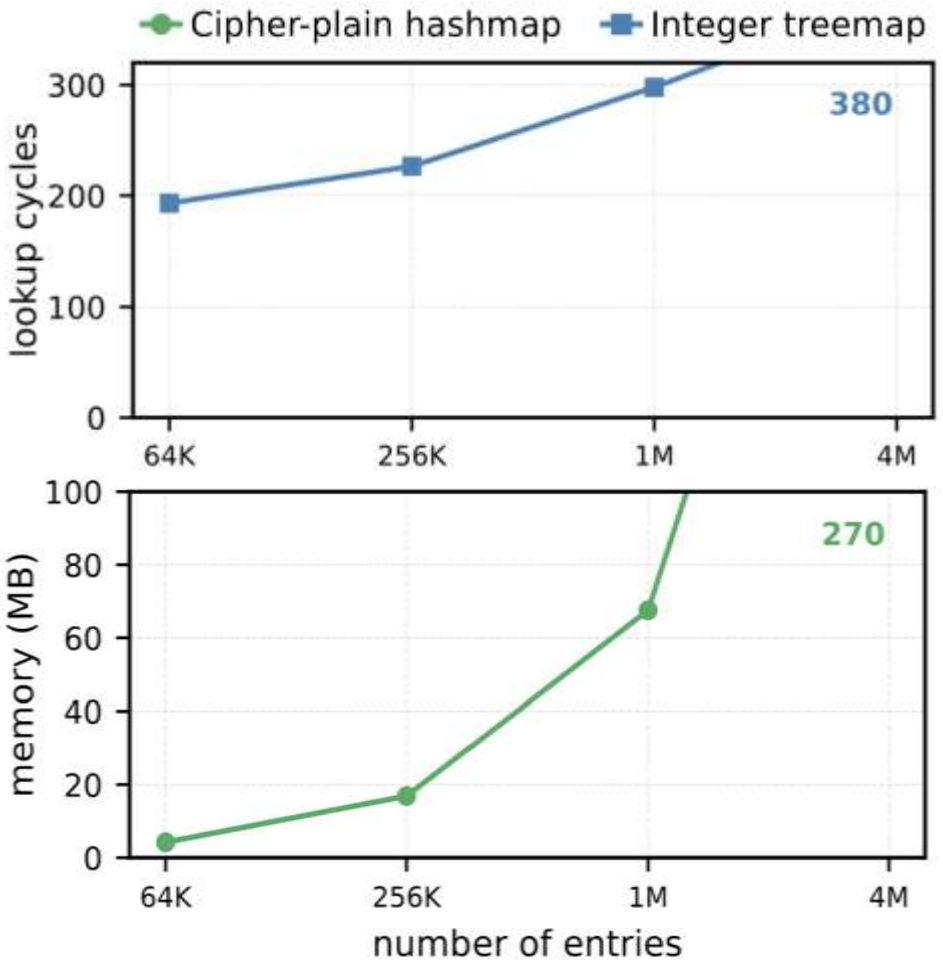


Level 1: Scaling in memory

- **Integer** FIDs: compact representation
- Data structure considerations:
  - ❌ Tree Maps: O(logN) lookup

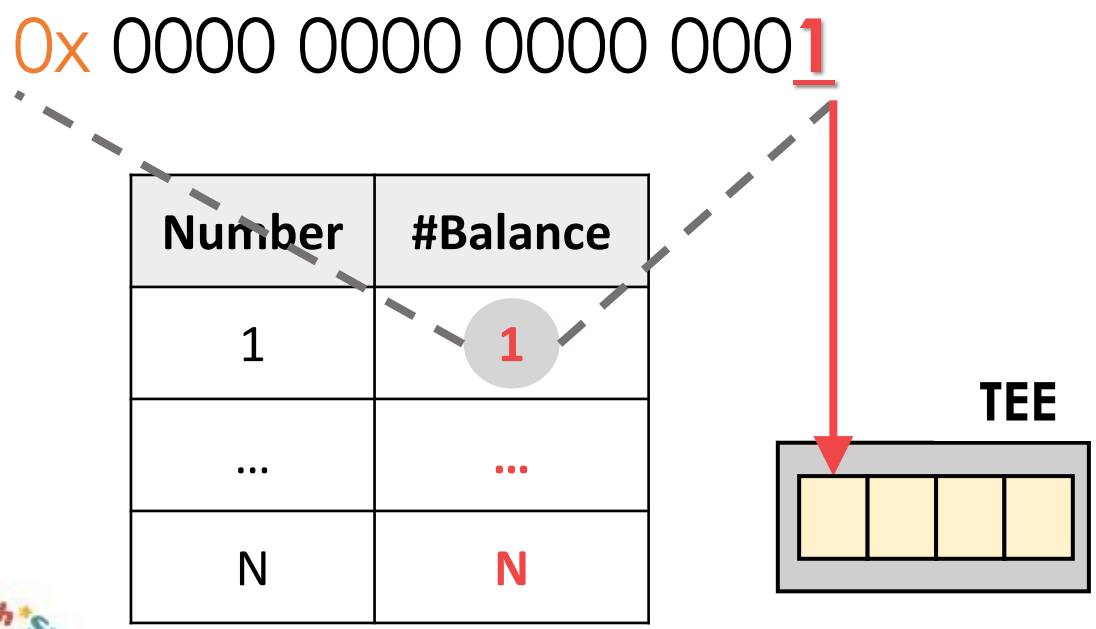
0x 0000 0000 0000 0001

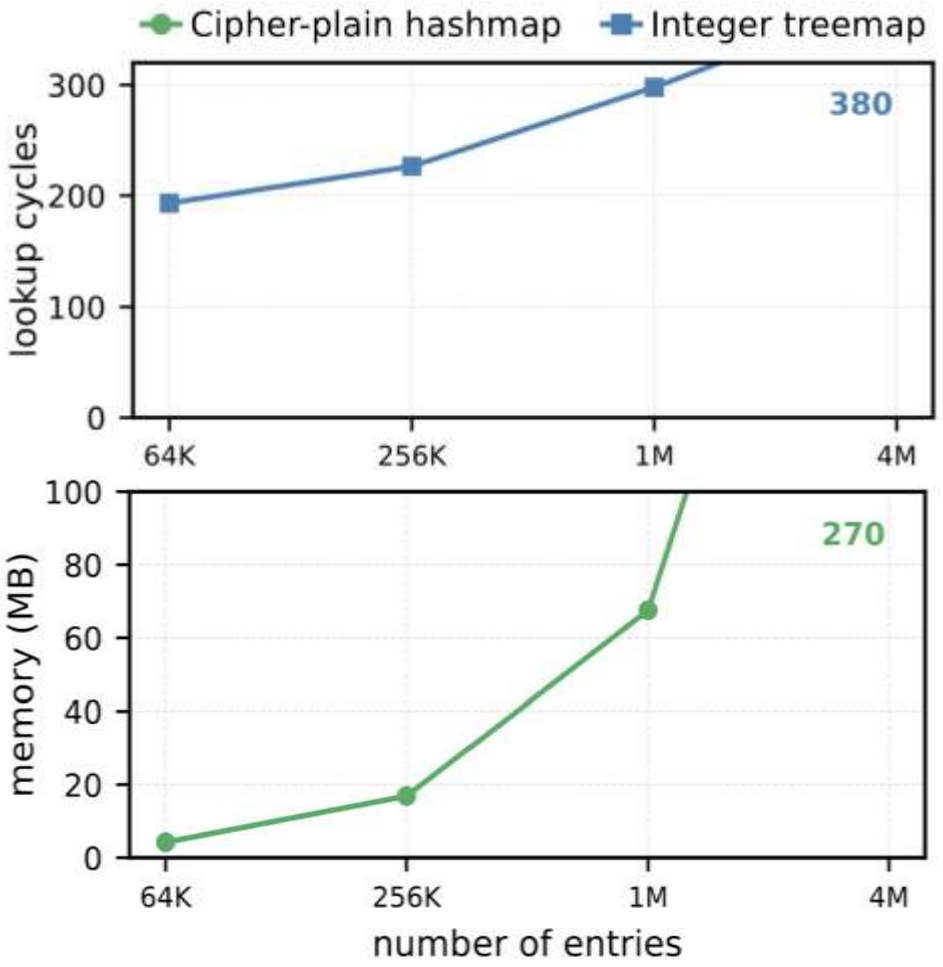
Number	#Balance
1	1
…	…
N	N



Level 1: Scaling in memory

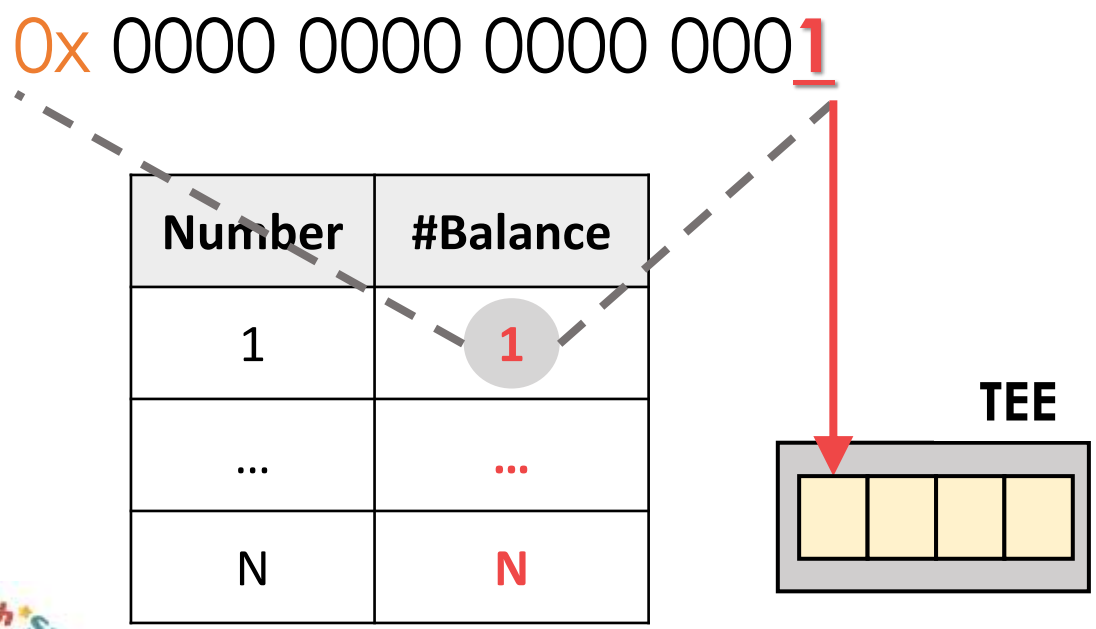
- **Integer** FIDs: compact representation
- Data structure considerations:
  - ❌ Tree Maps: O(logN) lookup
  - ✅ ZENO's Choice: Linear Array

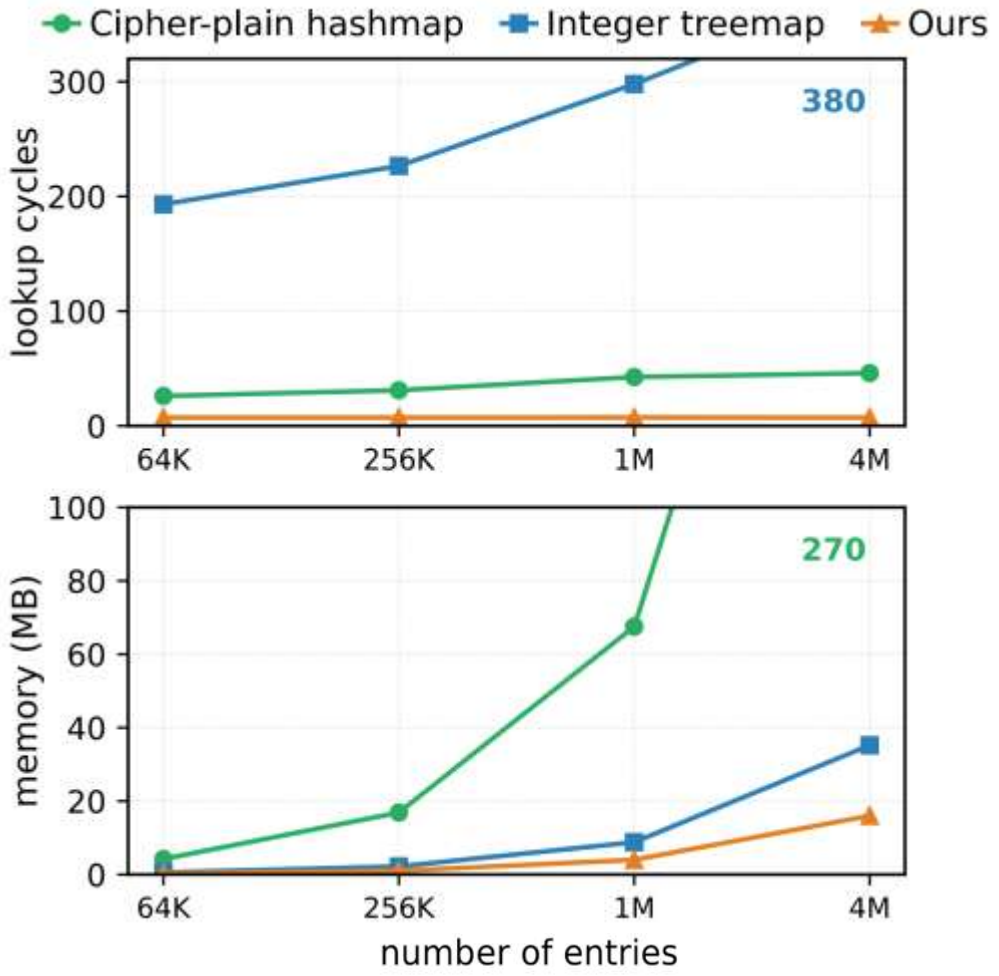




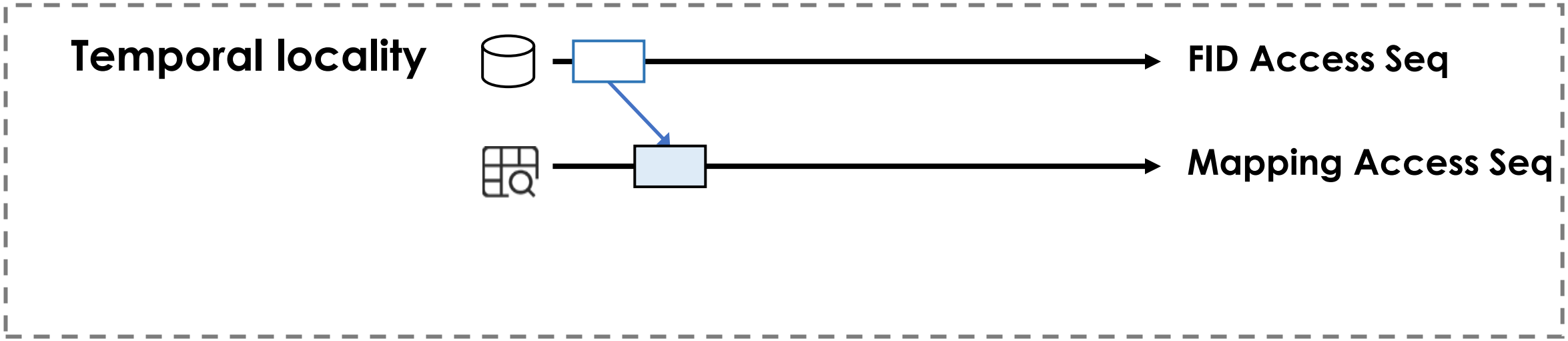
Level 1: Scaling in memory

- **Integer** FIDs: compact representation
- **Array-based** mapping store: O(1) direct indexing

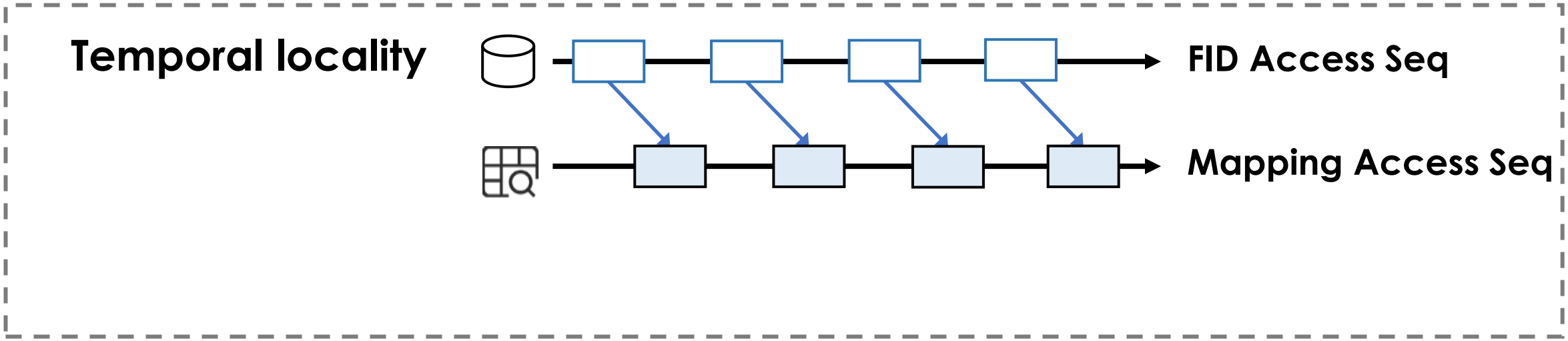




Level 2: Scaling to disk



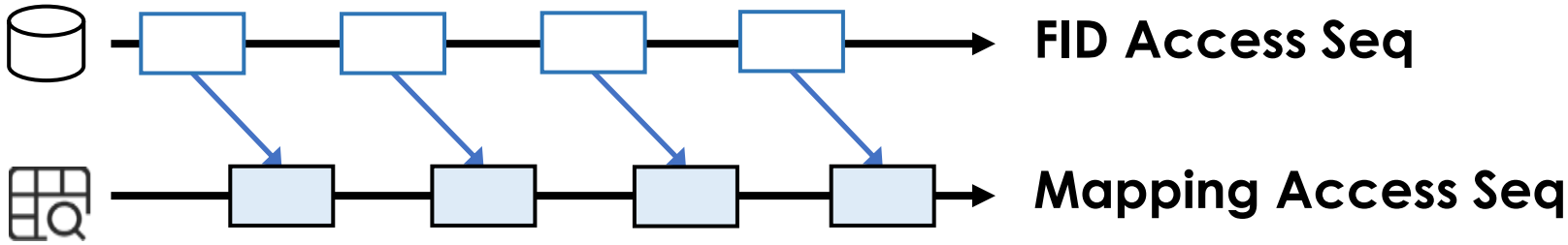
Level 2: Scaling to disk



Level 2: Scaling to disk

Temporal locality

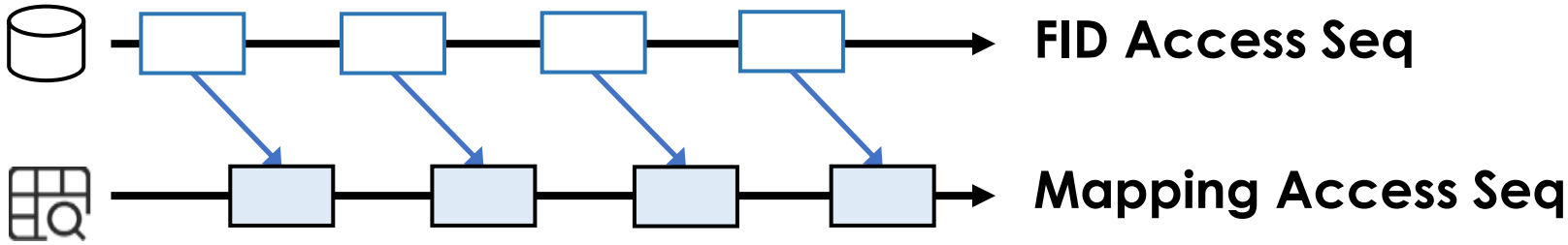
The DBMS drives field access order, and the MS simply follows.



Level 2: Scaling to disk

Temporal locality

The DBMS drives field access order, and the MS simply follows.

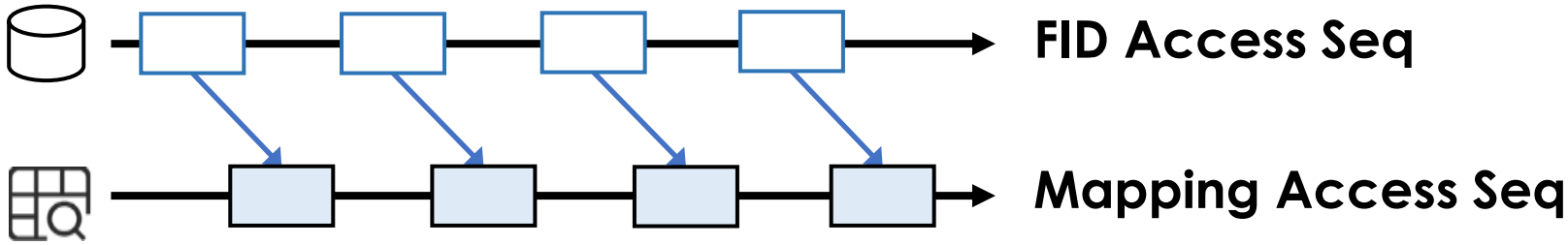


Bonus: Accesses to existing MS records are **lock-free**
*(Conflicts are already resolved by the **concurrency control** of DBMS)*

Level 2: Scaling to disk

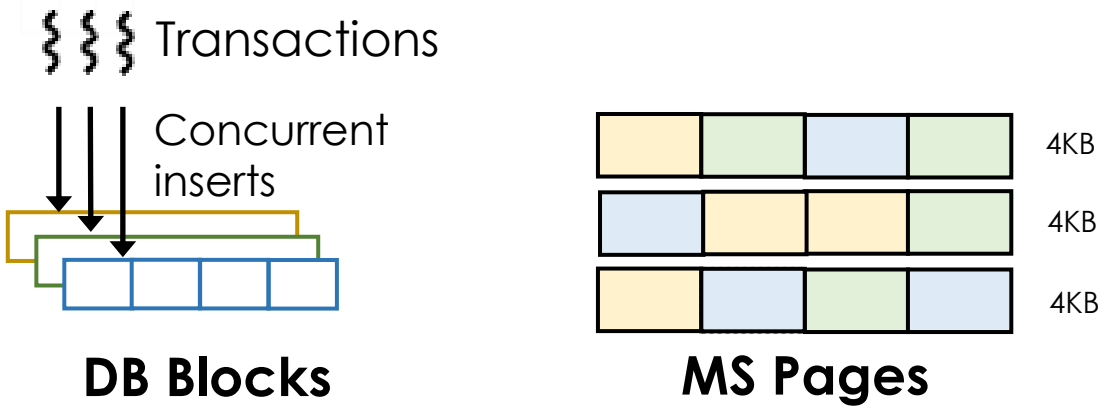
Temporal locality

The DBMS drives field access order, and the MS simply follows.



Bonus: Accesses to existing MS records are **lock-free**
*(Conflicts are already resolved by the **concurrency control** of DBMS)*

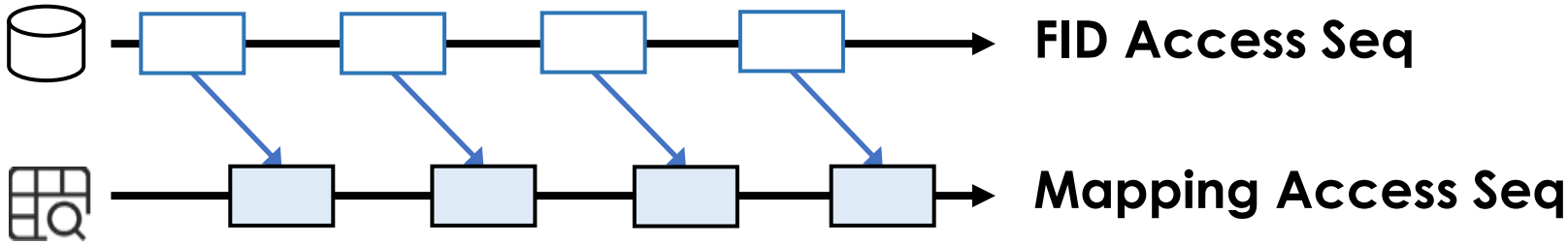
Spatial locality



Level 2: Scaling to disk

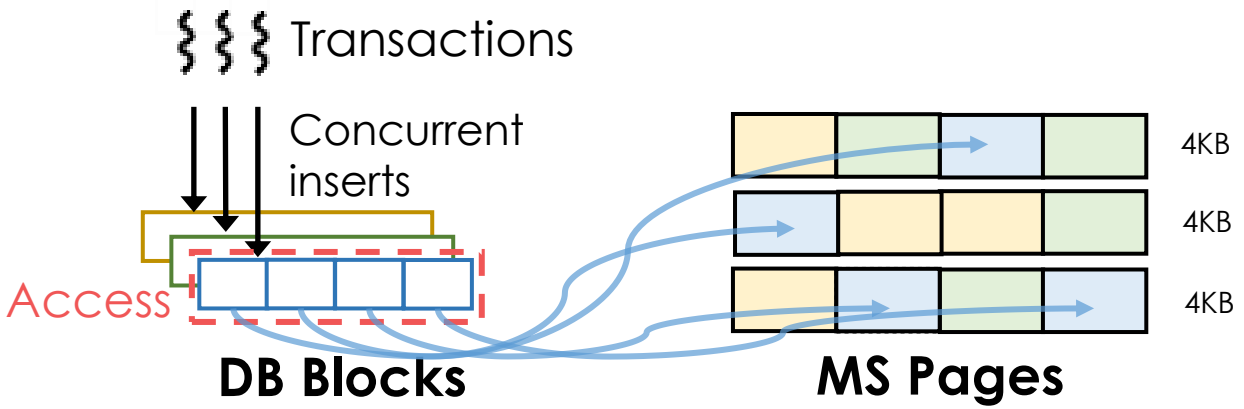
Temporal locality

The DBMS drives field access order, and the MS simply follows.



Bonus: Accesses to existing MS records are **lock-free**
*(Conflicts are already resolved by the **concurrency control** of DBMS)*

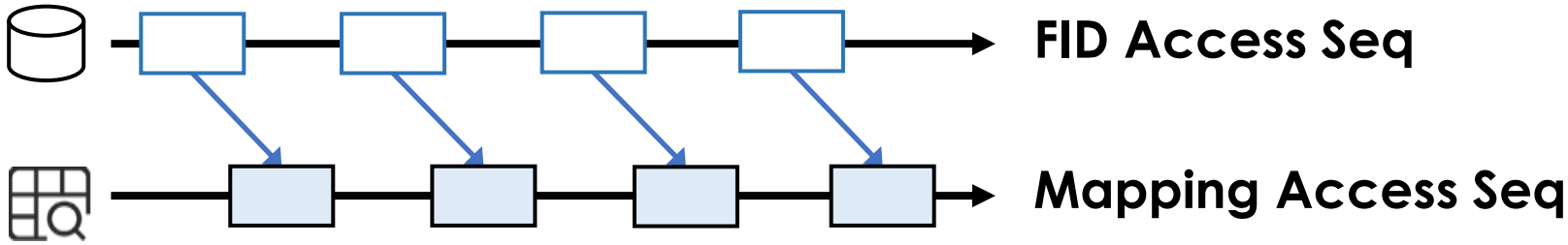
Spatial locality



Level 2: Scaling to disk

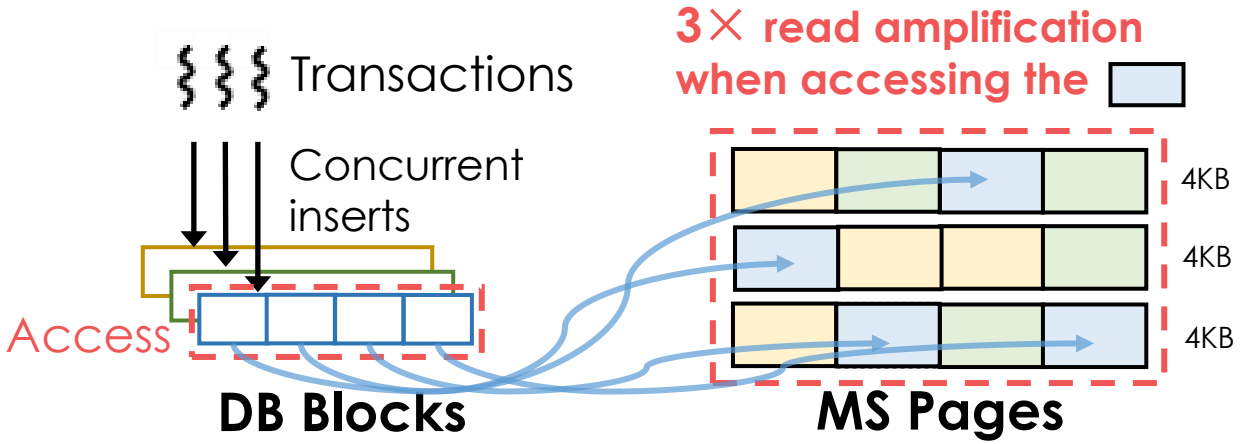
Temporal locality

The DBMS drives field access order, and the MS simply follows.



Bonus: Accesses to existing MS records are **lock-free**
*(Conflicts are already resolved by the **concurrency control** of DBMS)*

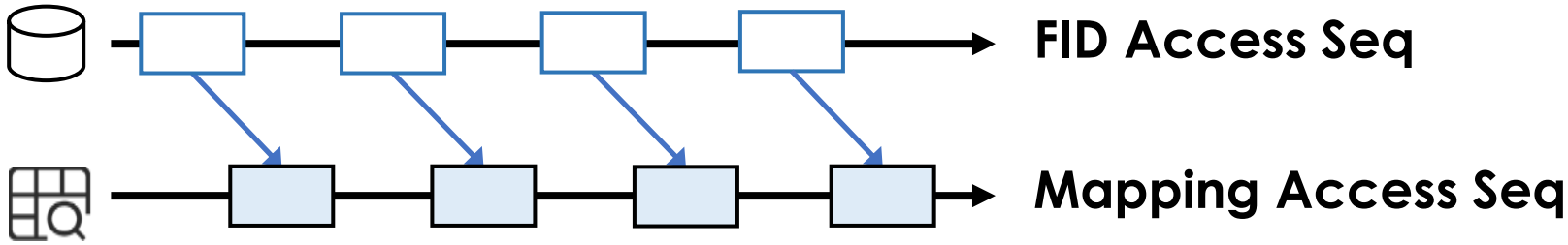
Spatial locality



Level 2: Scaling to disk

✅ Temporal locality

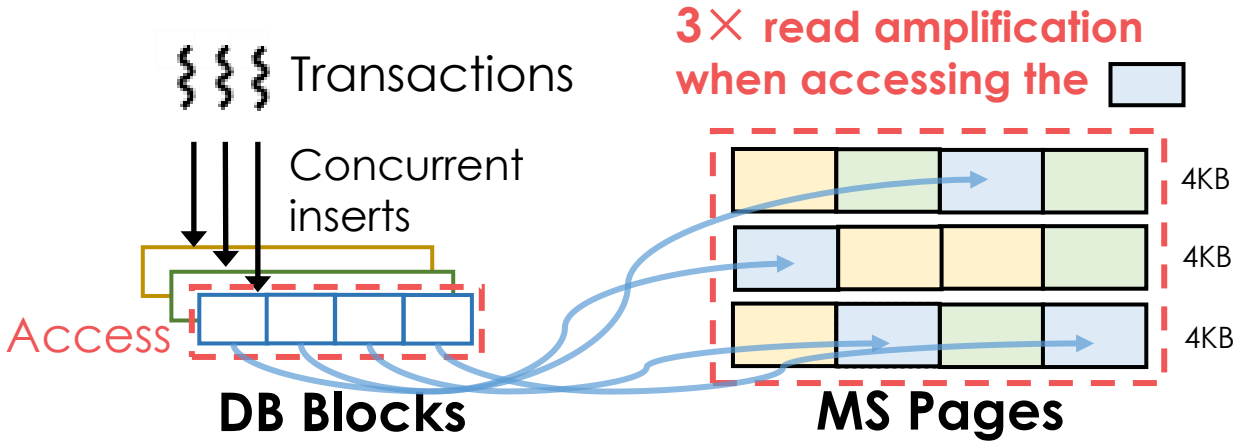
The DBMS drives field access order, and the MS simply follows.



Bonus: Accesses to existing MS records are **lock-free**
*(Conflicts are already resolved by the **concurrency control** of DBMS)*

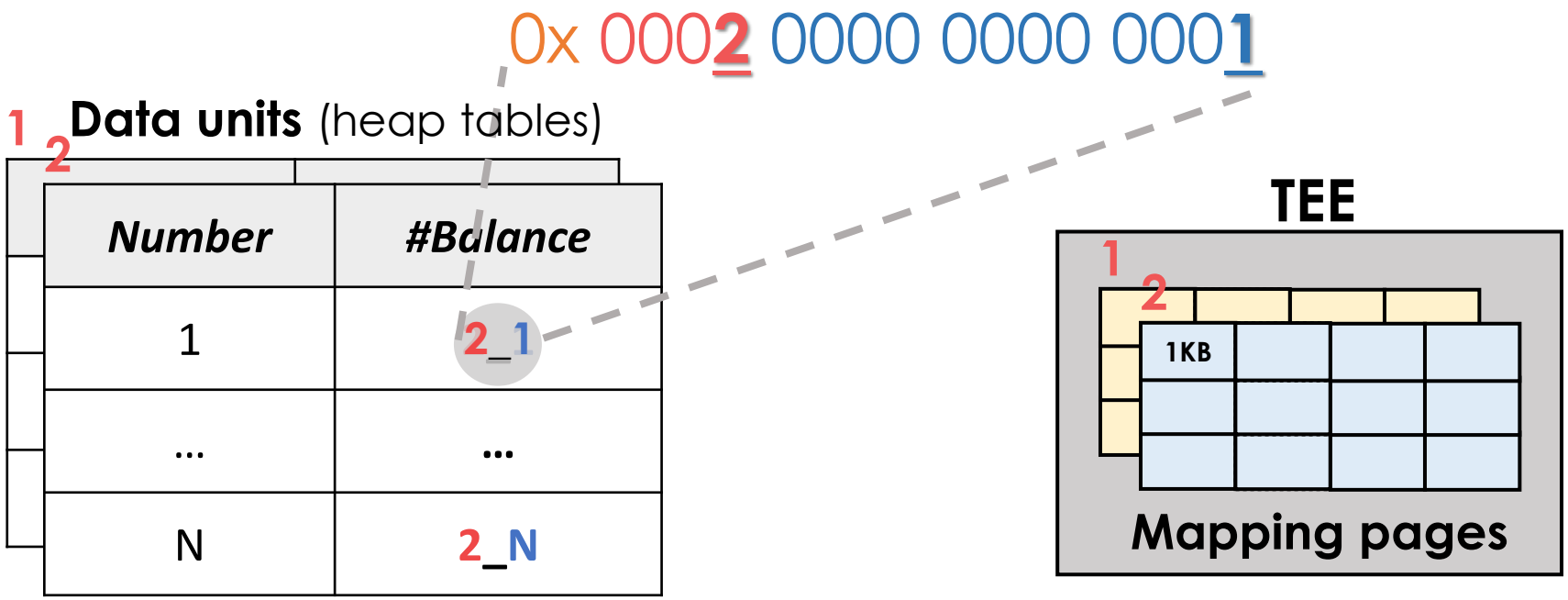
❌ Spatial locality

Root cause: **misalignment** between the storage layouts of the database and the mapping store.



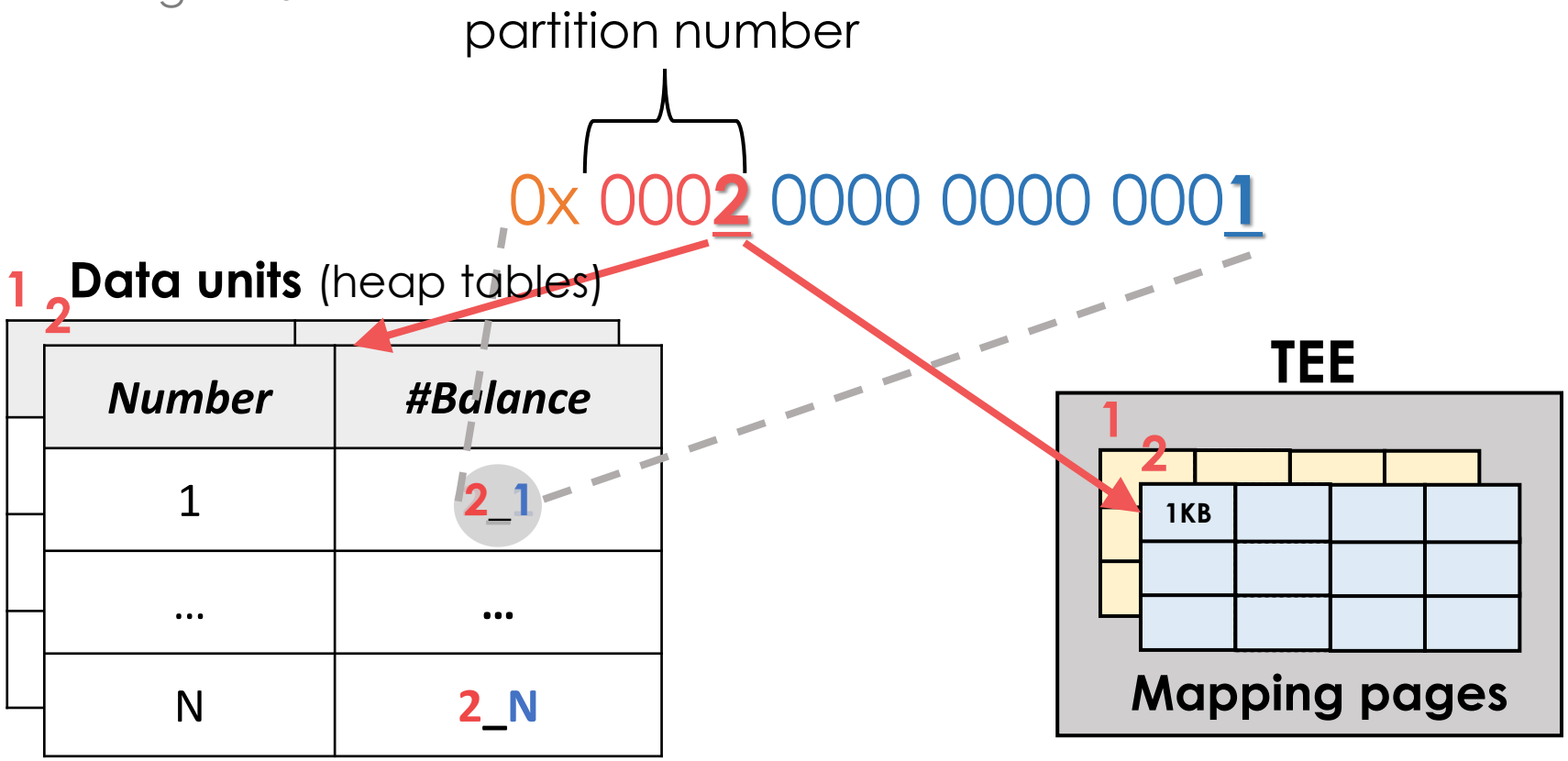
Partitioning by the DBMS storage layout

** Design shown for PostgreSQL*



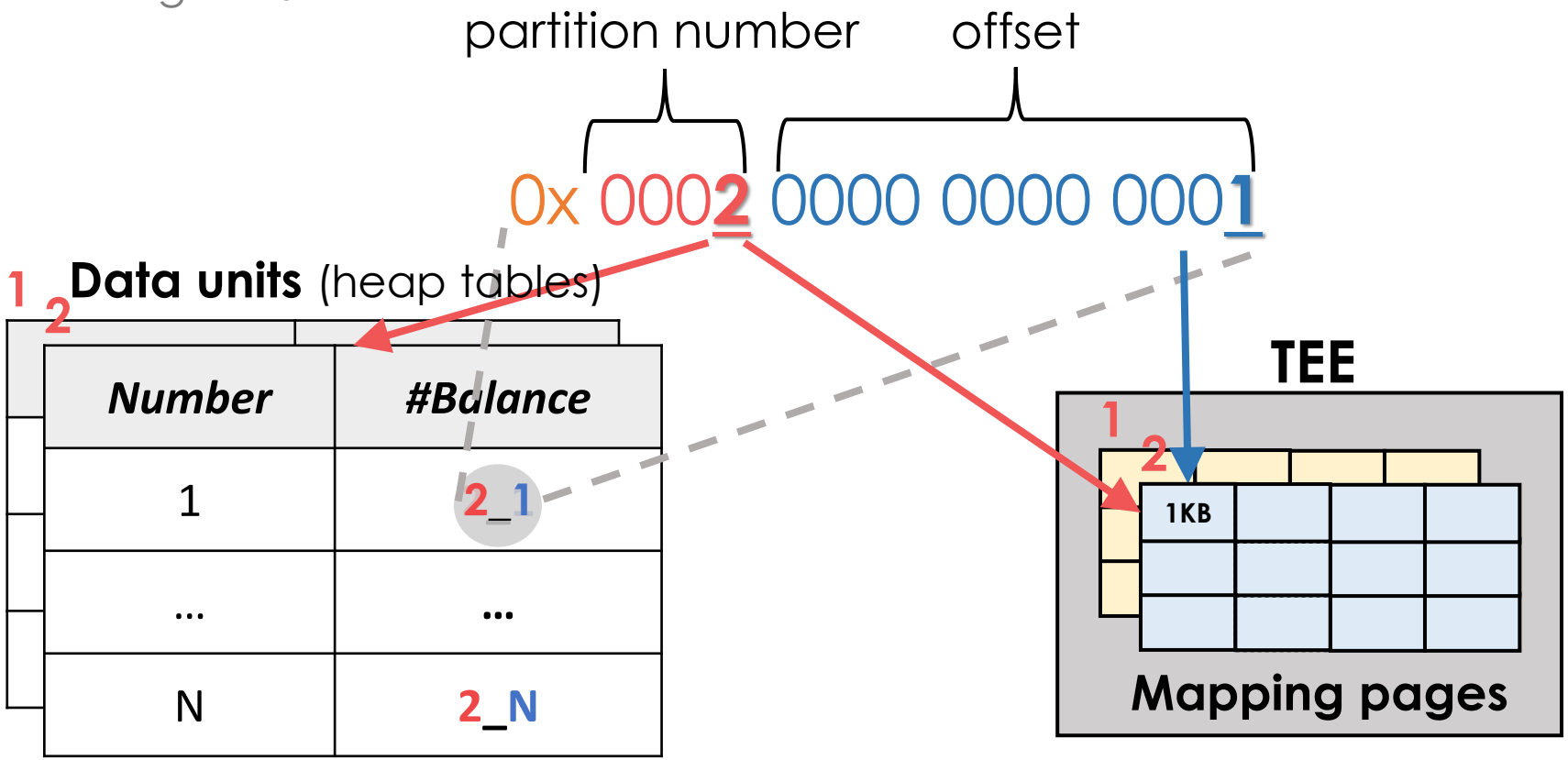
Partitioning by the DBMS storage layout

** Design shown for PostgreSQL*



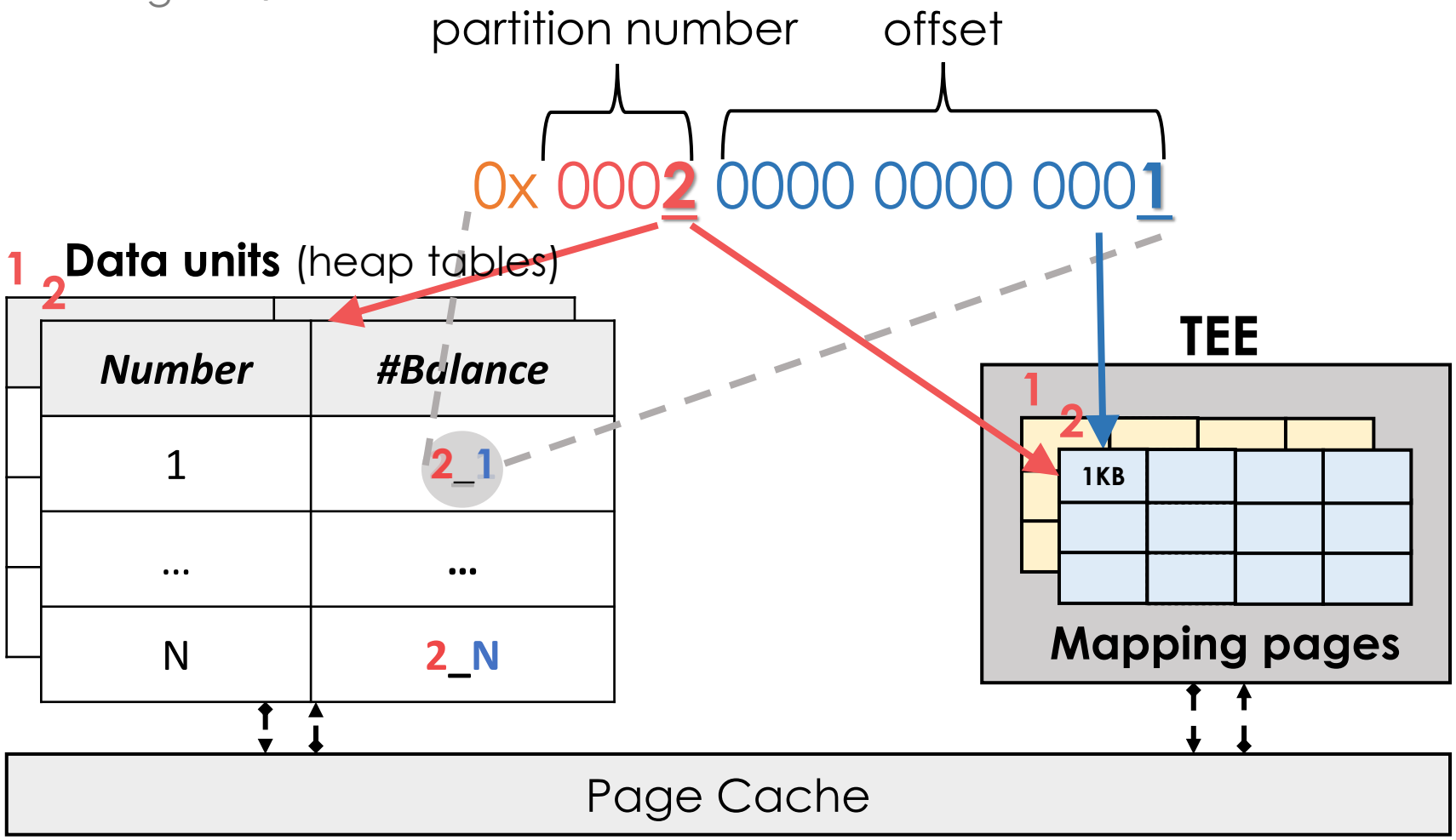
Partitioning by the DBMS storage layout

** Design shown for PostgreSQL*



Partitioning by the DBMS storage layout

** Design shown for PostgreSQL*



Sequential prefetching & cache replacement

Challenges

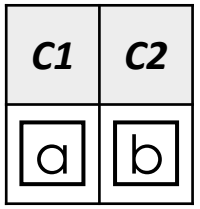
- #1: How to build a scalable mapping store?

- **#2: How to ensure transactional consistency between FIDs and their mappings?**

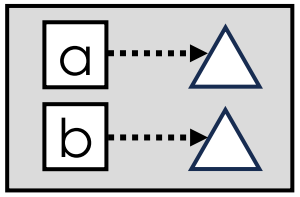
Crash scenarios during commit

Transaction S: INSERT INTO customer (balance) VALUES ('26');

DBMS Processing (parsing, executing…)



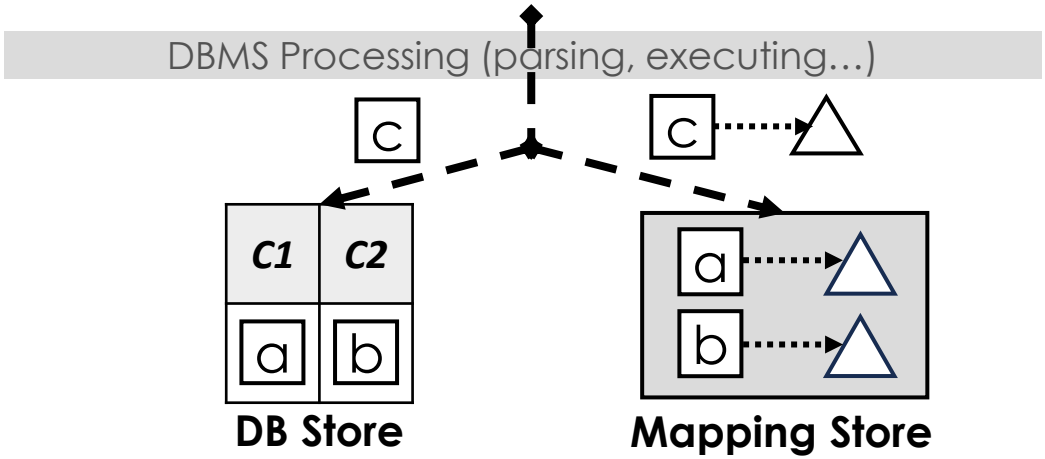
DB Store



Mapping Store

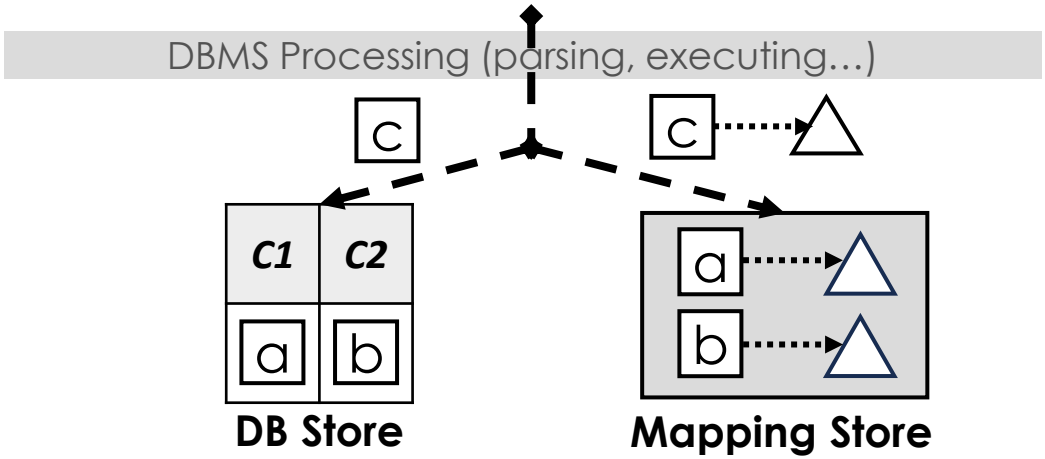
Crash scenarios during commit

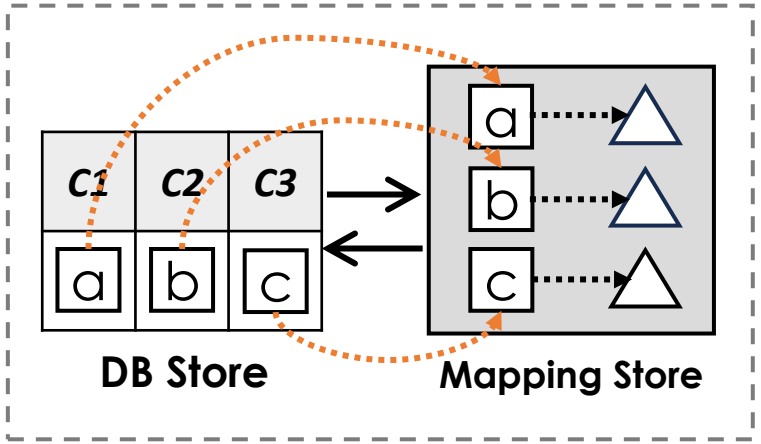
Transaction S: INSERT INTO customer (balance) VALUES ('26');



Crash scenarios during commit

Transaction S: INSERT INTO customer (balance) VALUES ('26');

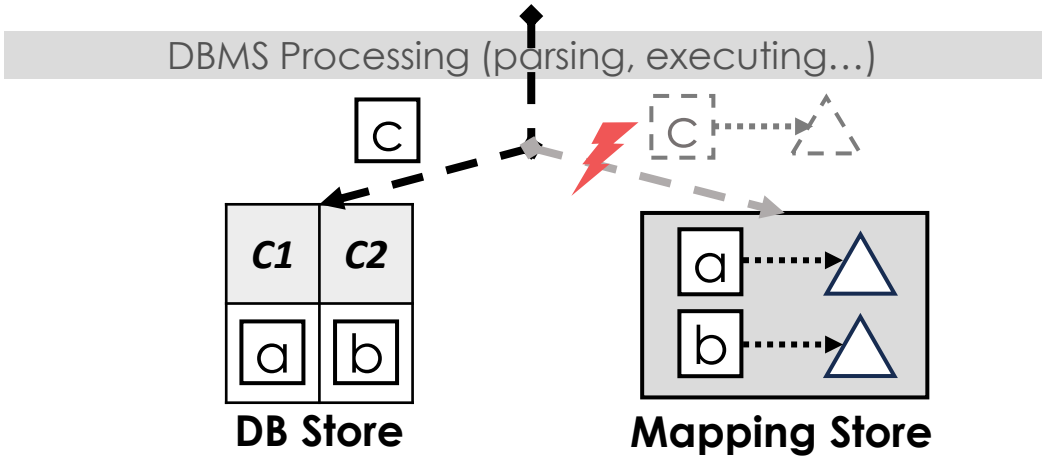


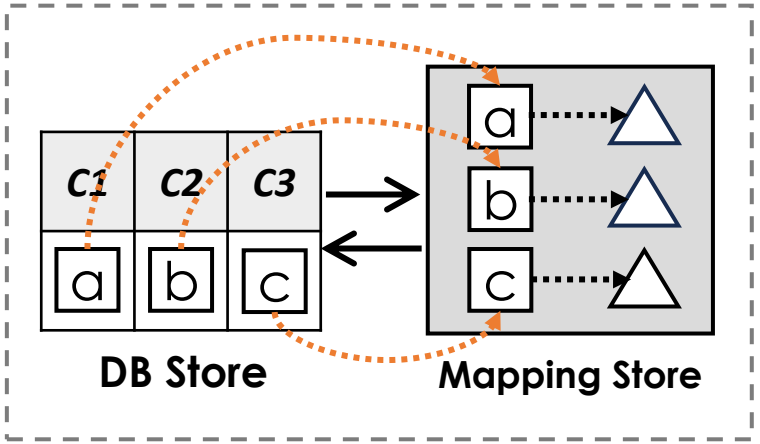


(a) Consistent state

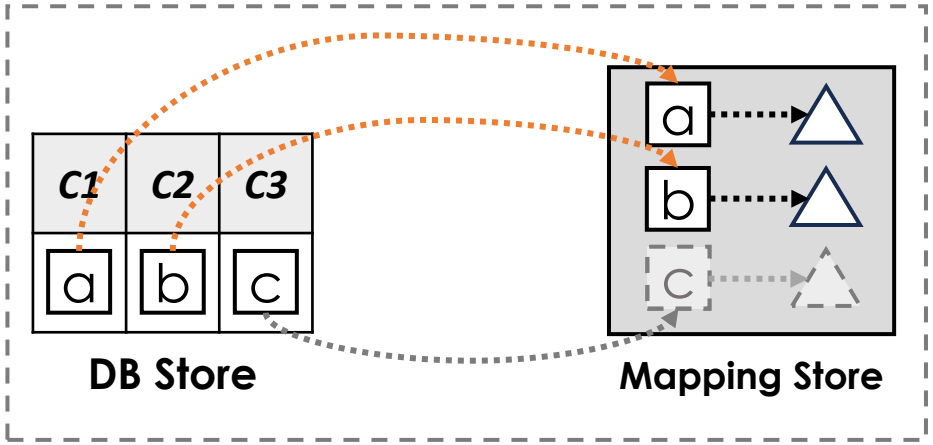
Crash scenarios during commit

Transaction S: INSERT INTO customer (balance) VALUES ('26');



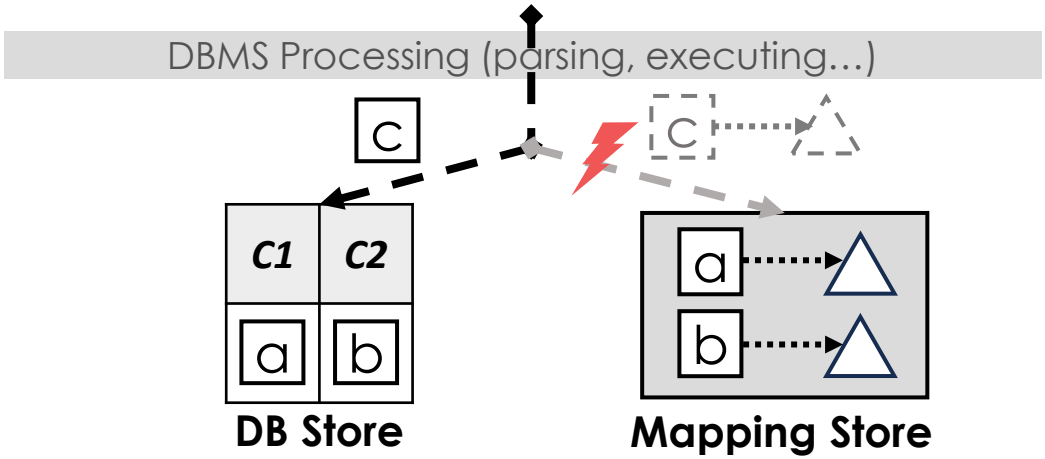


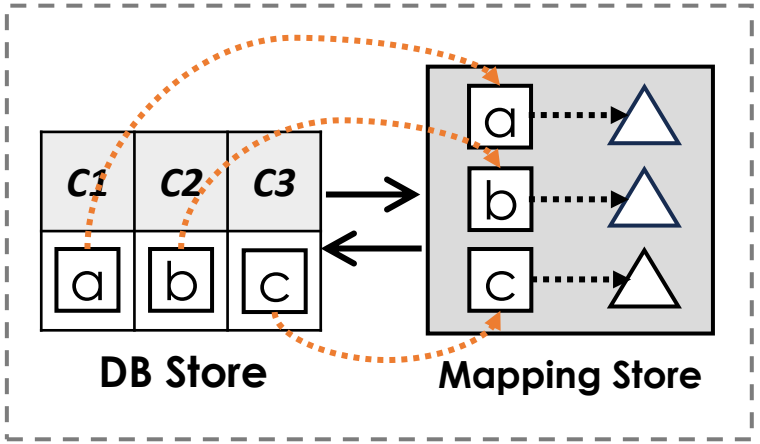
(a) Consistent state



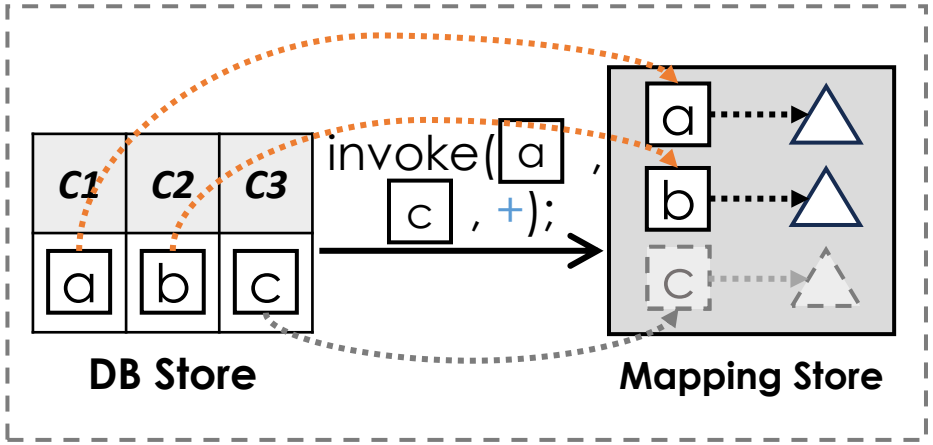
Crash scenarios during commit

Transaction S: INSERT INTO customer (balance) VALUES ('26');



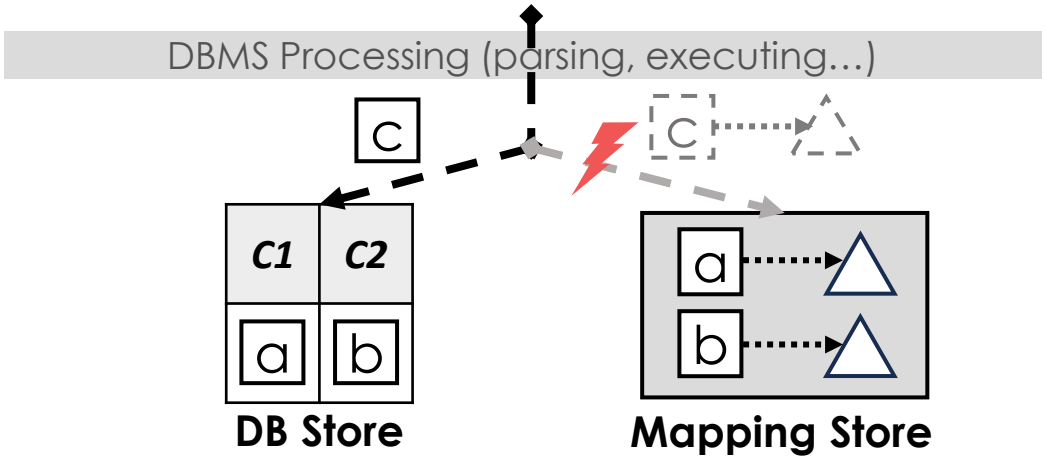


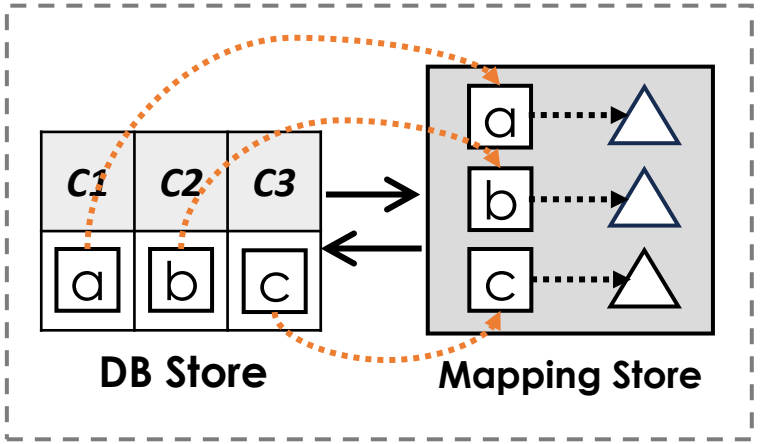
(a) Consistent state



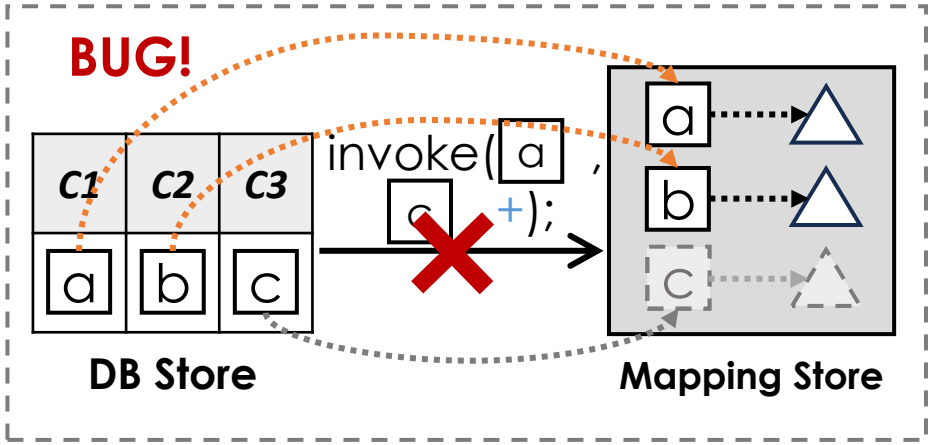
Crash scenarios during commit

Transaction S: INSERT INTO customer (balance) VALUES ('26');





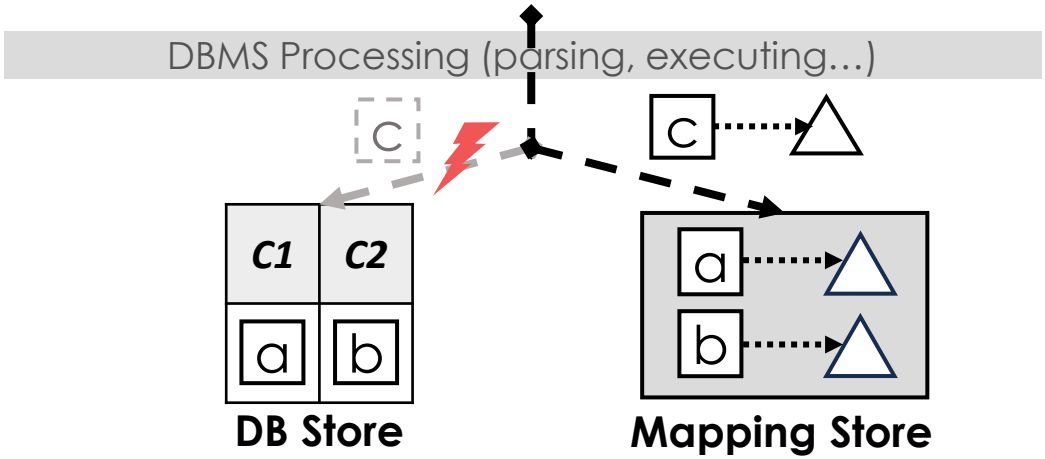
(a) Consistent state

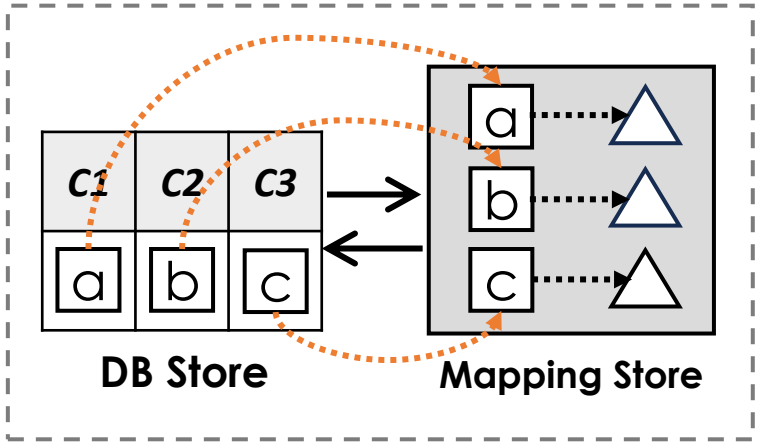


(b) Missing-mapping state

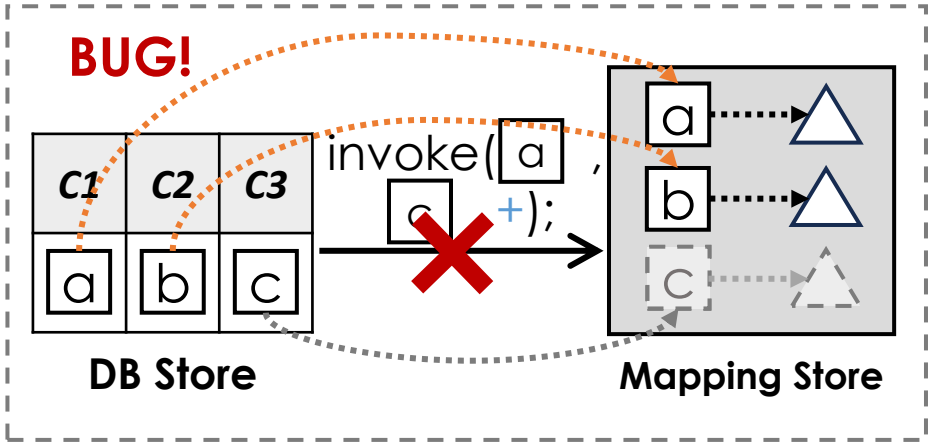
Crash scenarios during commit

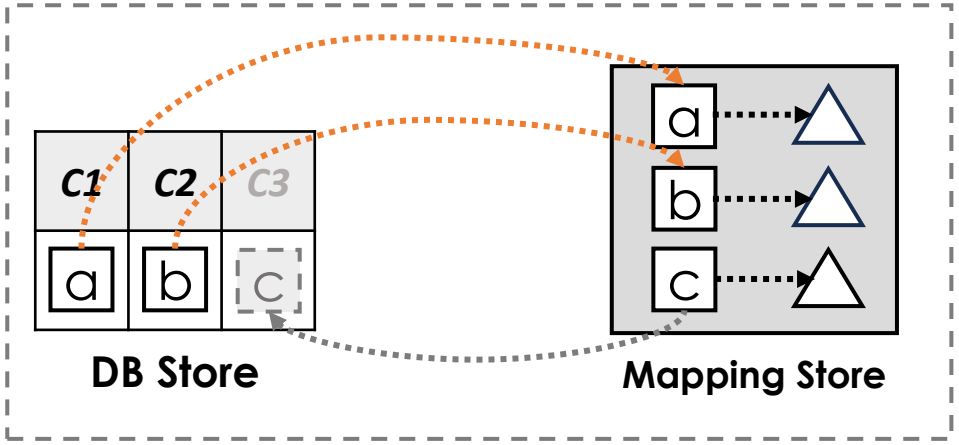
Transaction S: INSERT INTO customer (balance) VALUES ('26');





(a) Consistent state

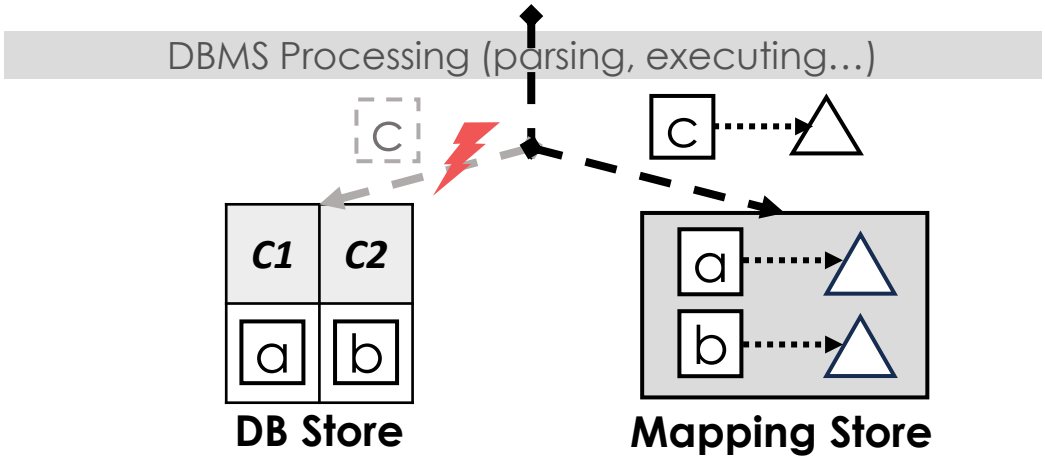


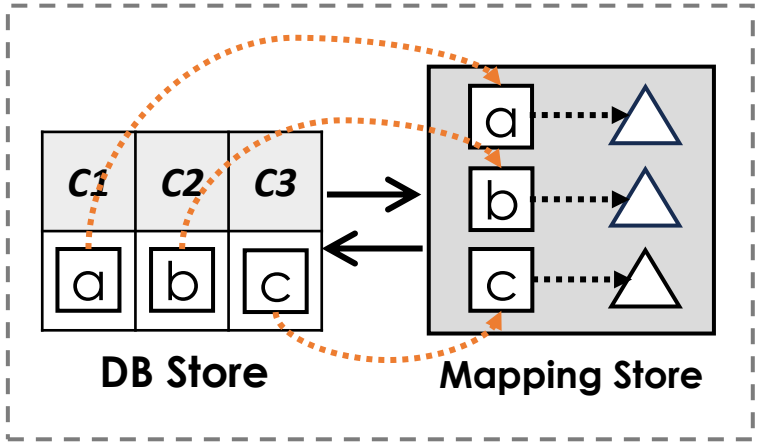


(b) Missing-mapping state

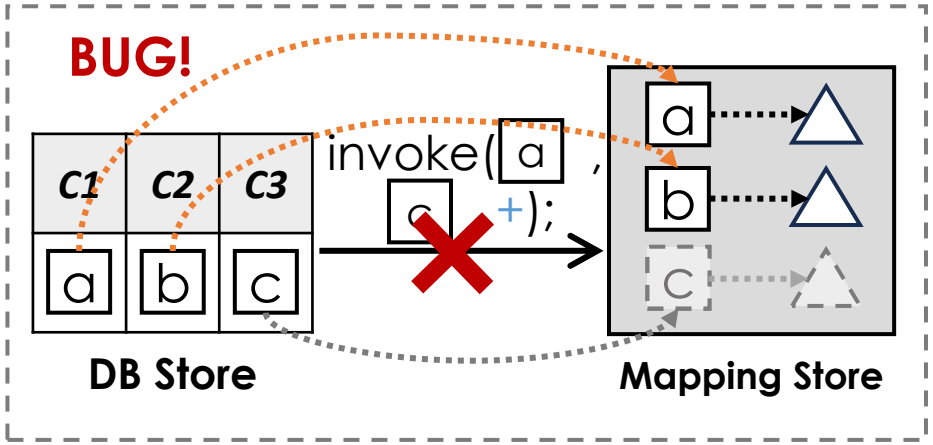
Crash scenarios during commit

Transaction S: INSERT INTO customer (balance) VALUES ('26');

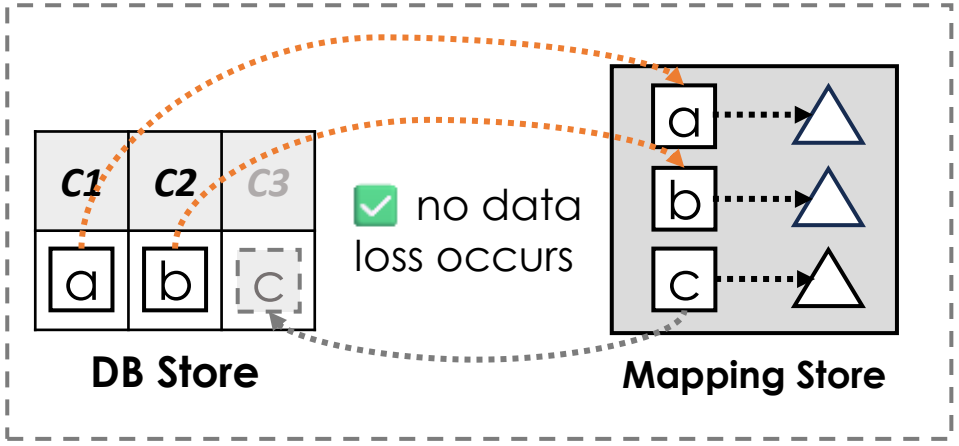




(a) Consistent state



(b) Missing-mapping state

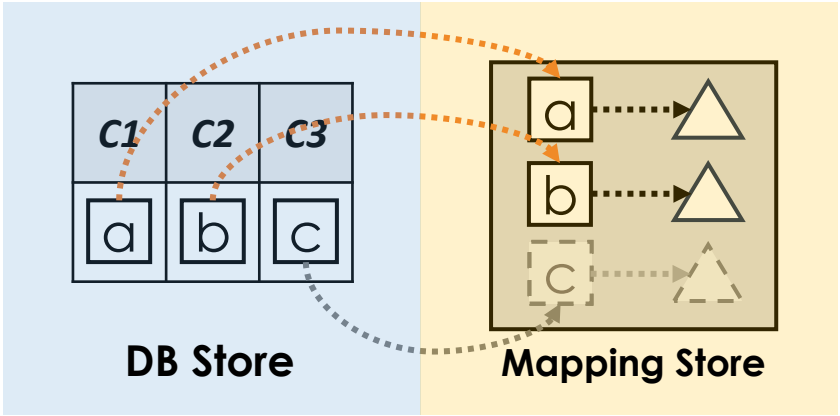


(c) Orphan-mapping state

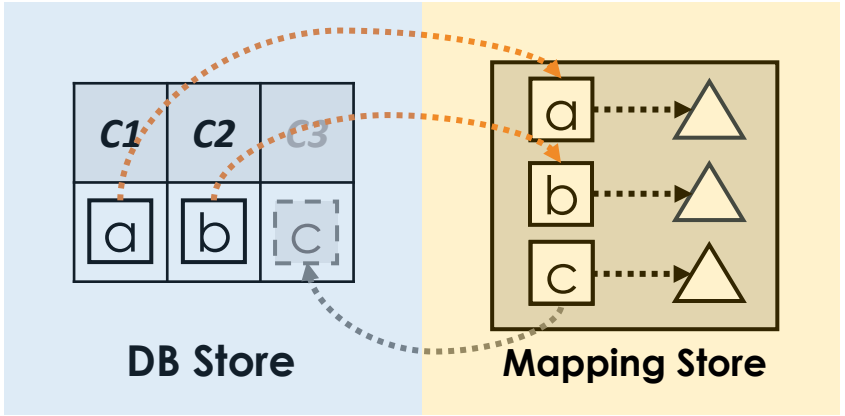
Observation: one-way consistency

Every DBMS-**visible** FID cannot miss a valid mapping (b).

Invisible orphan mappings are acceptable (c).



(b) Missing-mapping state



(c) Orphan-mapping state

Ensuring crash consistency

MS Mappings MUST be durable before DB FIDs.

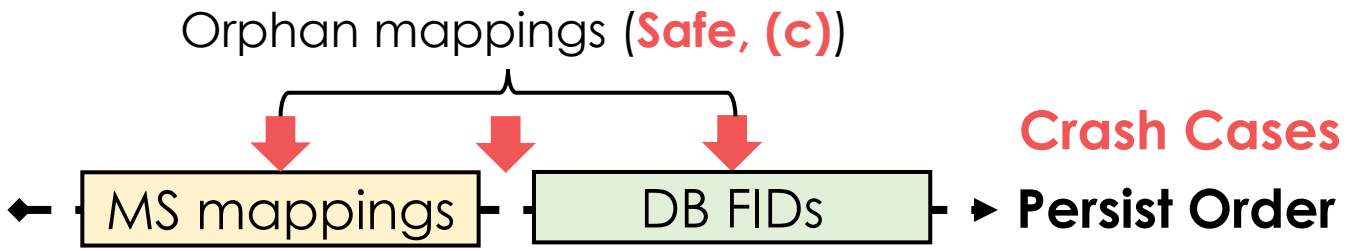
Persistence Ordering



Ensuring crash consistency

MS Mappings MUST be durable before DB FIDs.

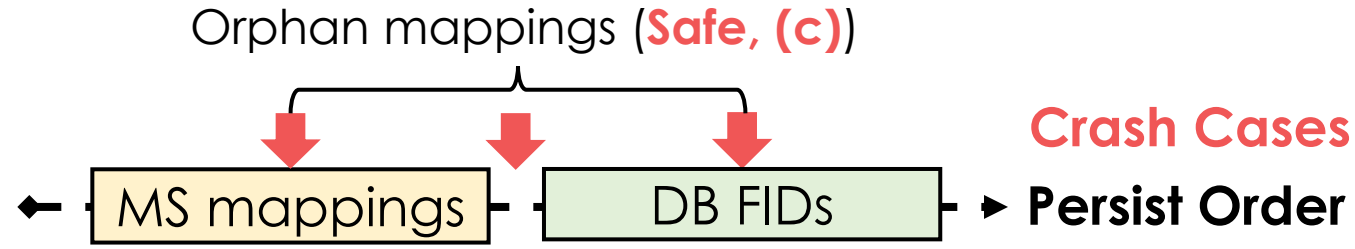
Persistence Ordering



Ensuring crash consistency

MS Mappings MUST be durable before DB FIDs.

Persistence Ordering

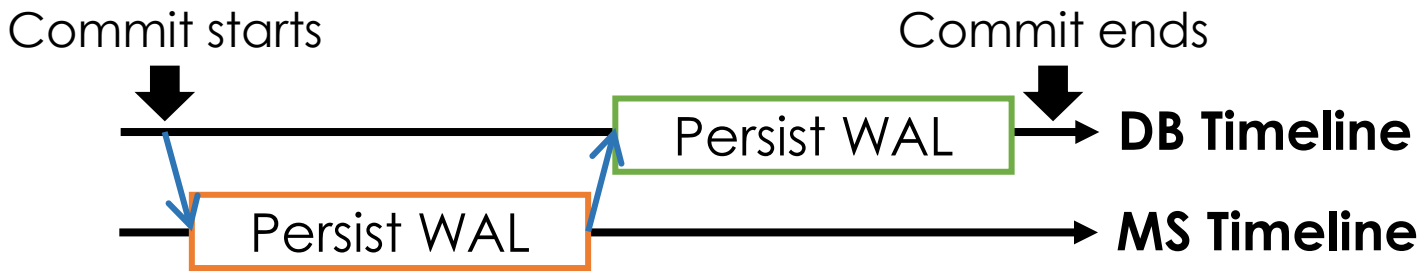


Persistence ordering

* *Write-Ahead Log*

→ WAL ordering

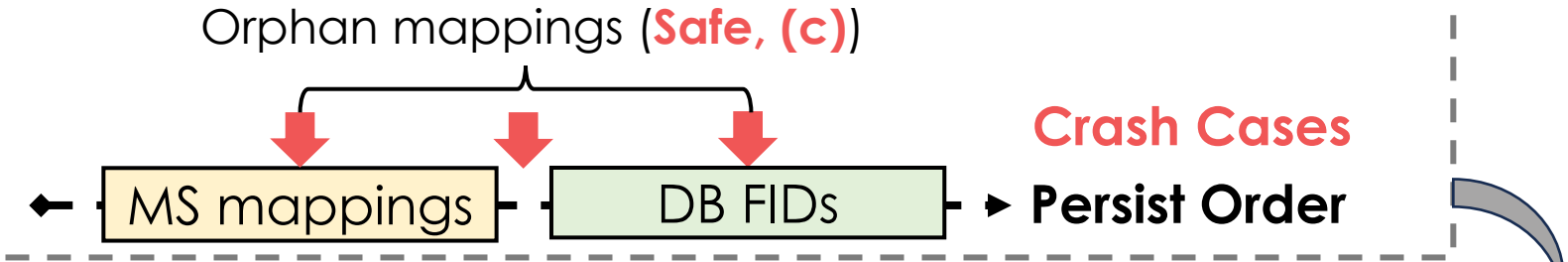
Commit Protocol



Ensuring crash consistency

MS Mappings MUST be durable before DB FIDs.

Persistence Ordering



Persistence ordering

* *Write-Ahead Log*

→ WAL ordering

Commit Protocol

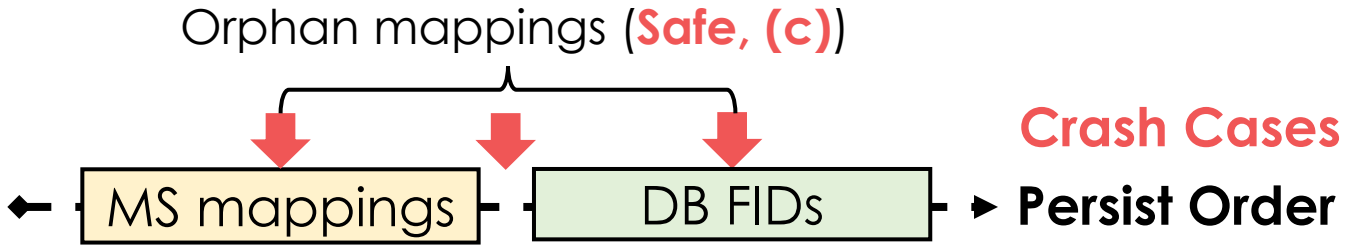


Two serialized WAL writes?

Ensuring crash consistency

MS Mappings MUST be durable before DB FIDs.

Persistence Ordering

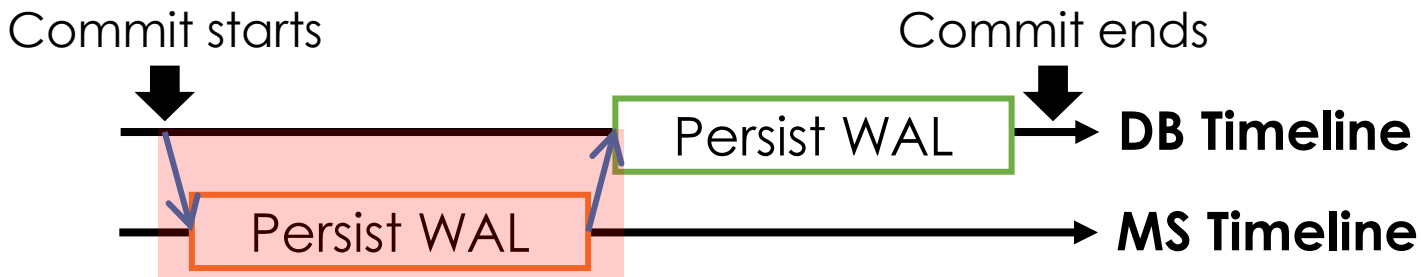


Persistence ordering

* *Write-Ahead Log*

→ WAL ordering

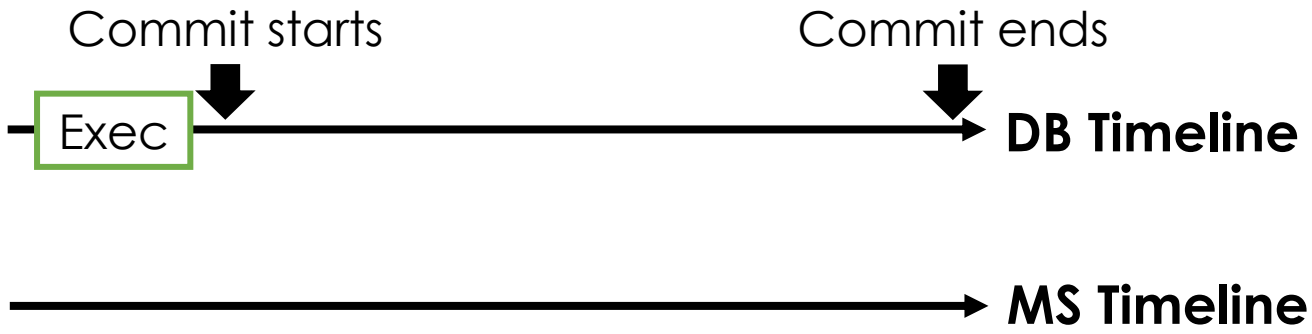
Commit Protocol



Two serialized WAL writes?

Merge into a single, co-designed WAL stream.

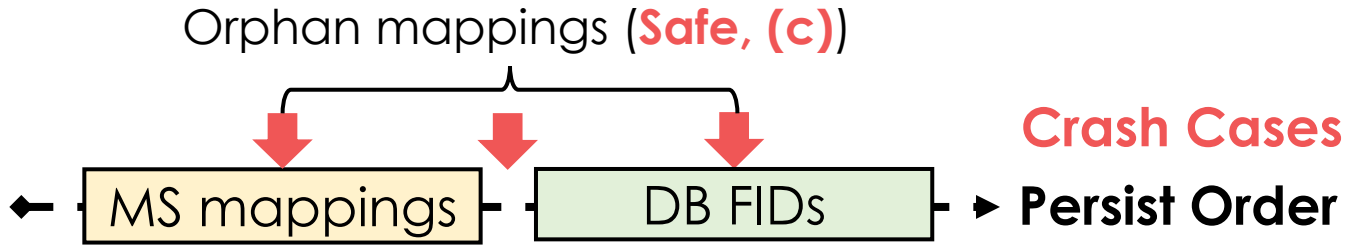
Co-designed WAL



Ensuring crash consistency

MS Mappings MUST be durable before DB FIDs.

Persistence Ordering

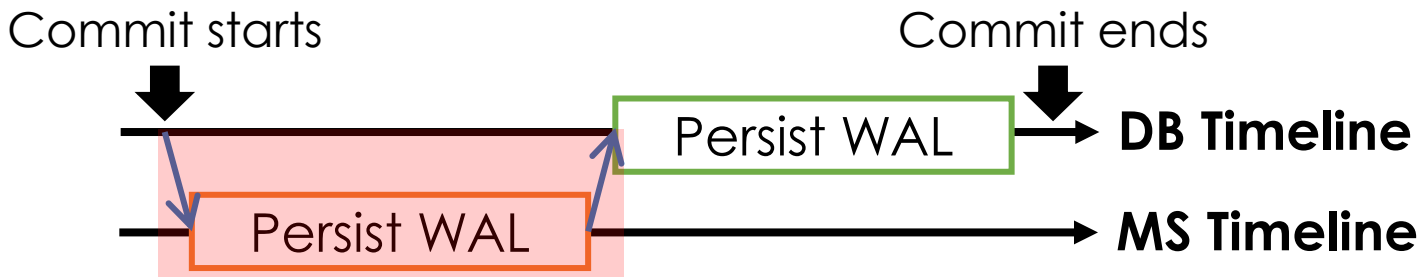


Persistence ordering

* *Write-Ahead Log*

→ WAL ordering

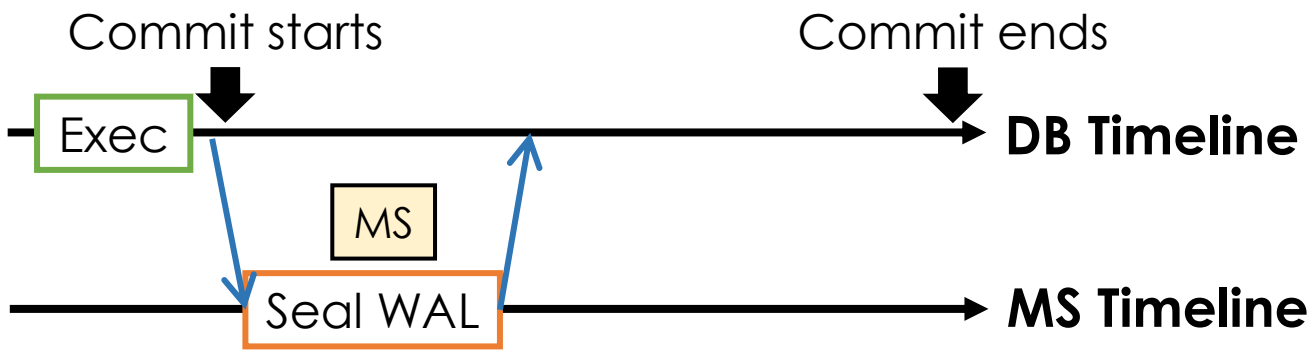
Commit Protocol



Two serialized WAL writes?

Merge into a single, co-designed WAL stream.

Co-designed WAL

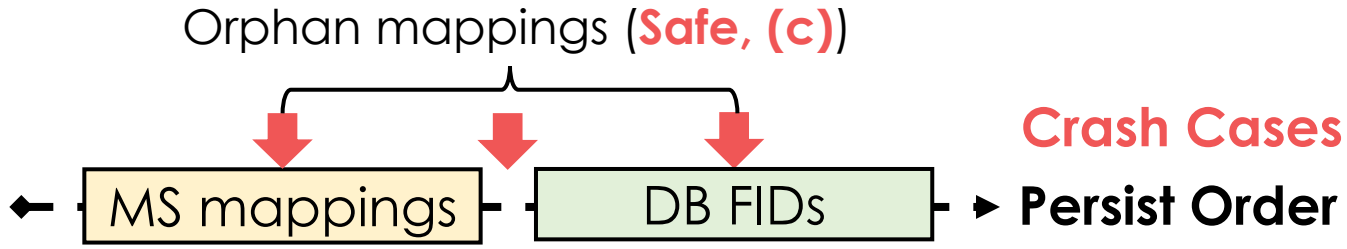


*Seal: Encrypt & copy to untrusted memory

Ensuring crash consistency

MS Mappings MUST be durable before DB FIDs.

Persistence Ordering

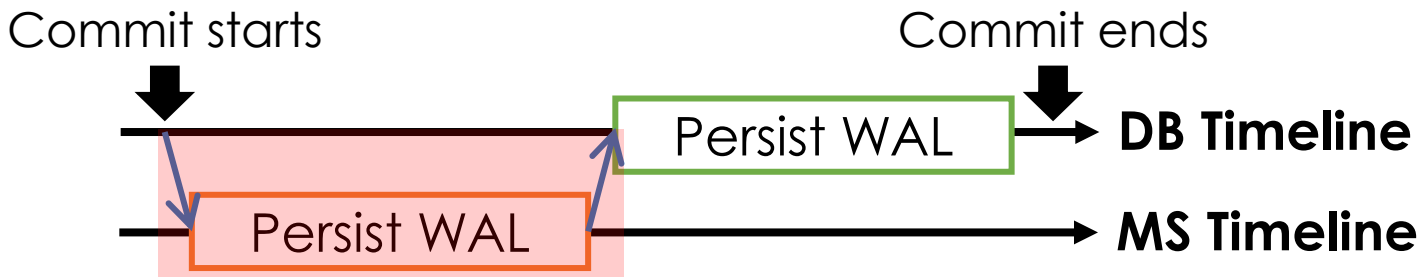


Persistence ordering

* *Write-Ahead Log*

→ WAL ordering

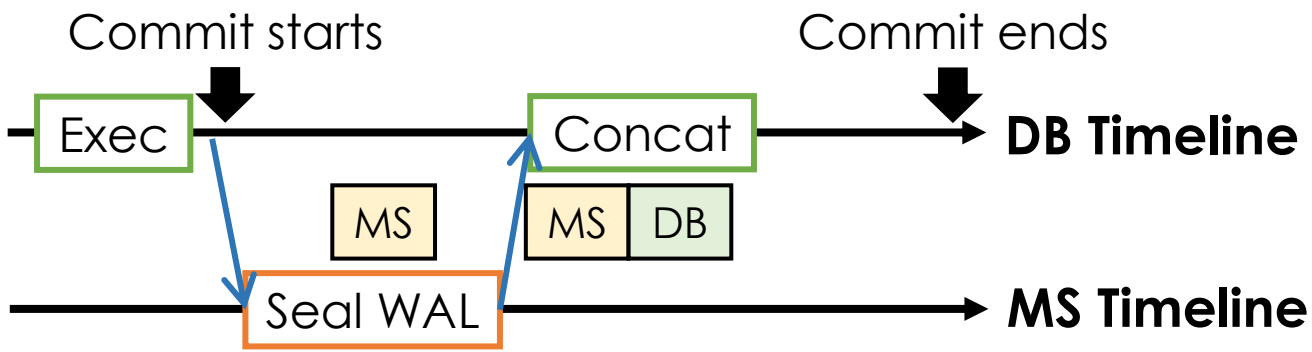
Commit Protocol



Two serialized WAL writes?

Merge into a single, co-designed WAL stream.

Co-designed WAL

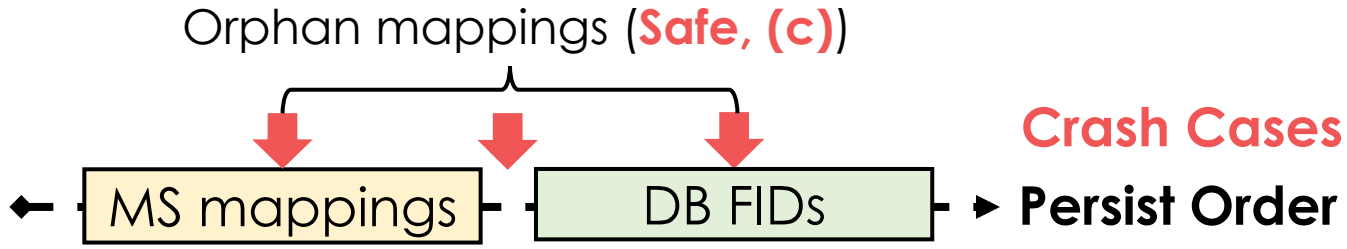


*Seal: Encrypt & copy to untrusted memory

Ensuring crash consistency

MS Mappings MUST be durable before DB FIDs.

Persistence Ordering

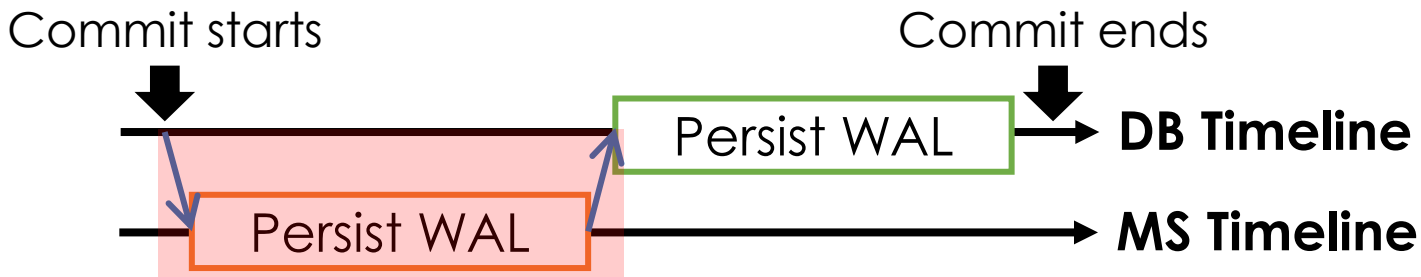


Persistence ordering

* ***W****rite-****A****head* ***L****og*

→ WAL ordering

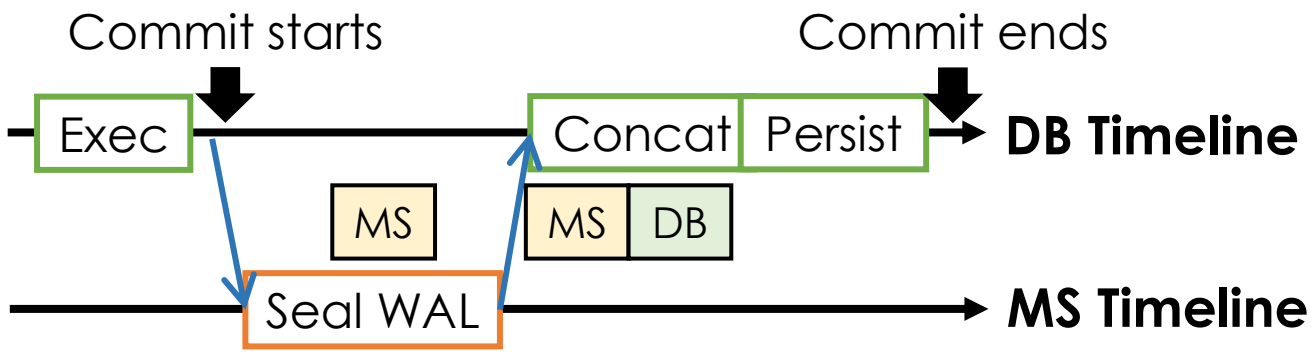
Commit Protocol



Two serialized WAL writes?

Merge into a single, co-designed WAL stream.

Co-designed WAL

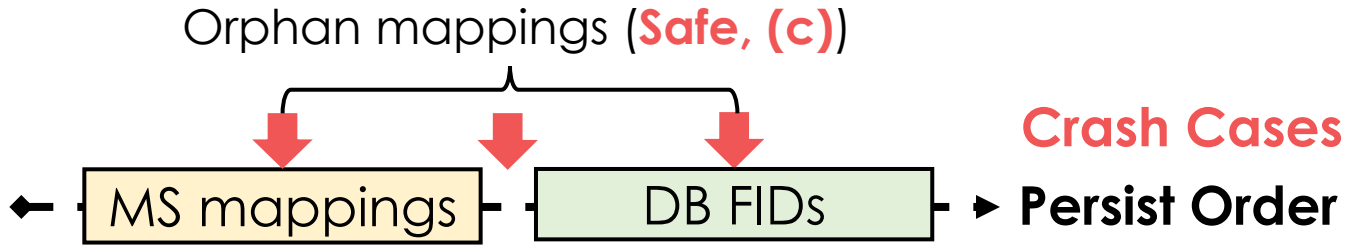


*Seal: Encrypt & copy to untrusted memory

Ensuring crash consistency

MS Mappings MUST be durable before DB FIDs.

Persistence Ordering

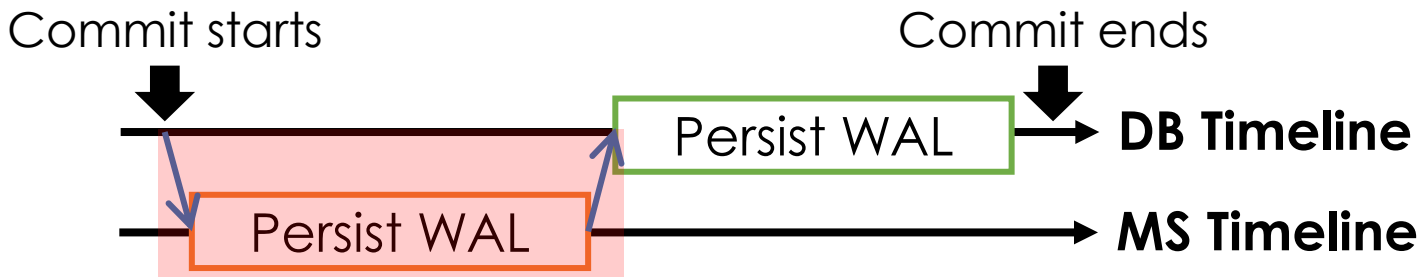


Persistence ordering

* *Write-Ahead Log*

→ WAL ordering

Commit Protocol

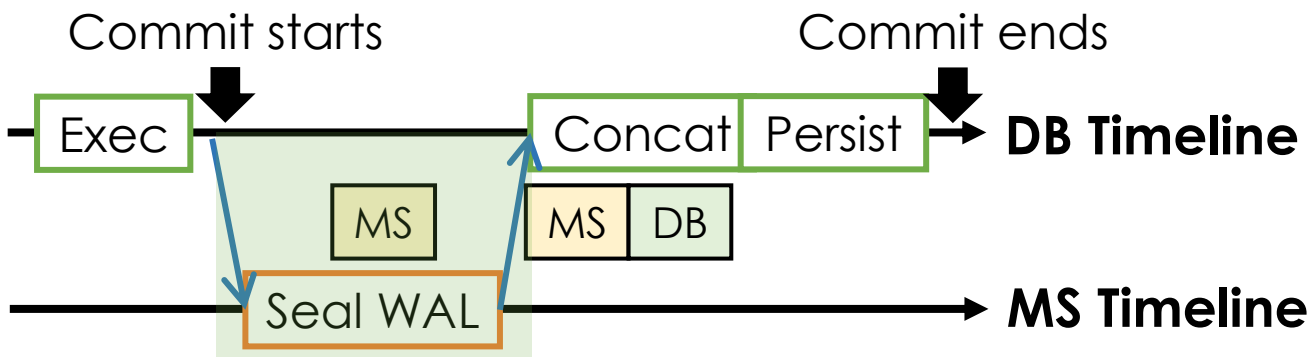


Two serialized WAL writes?

Merge into a single, co-designed WAL stream.

~~Serialized I/O~~ → **~1-Block Enc**

Co-designed WAL



*Seal: Encrypt & copy to untrusted memory

Other techniques and analysis

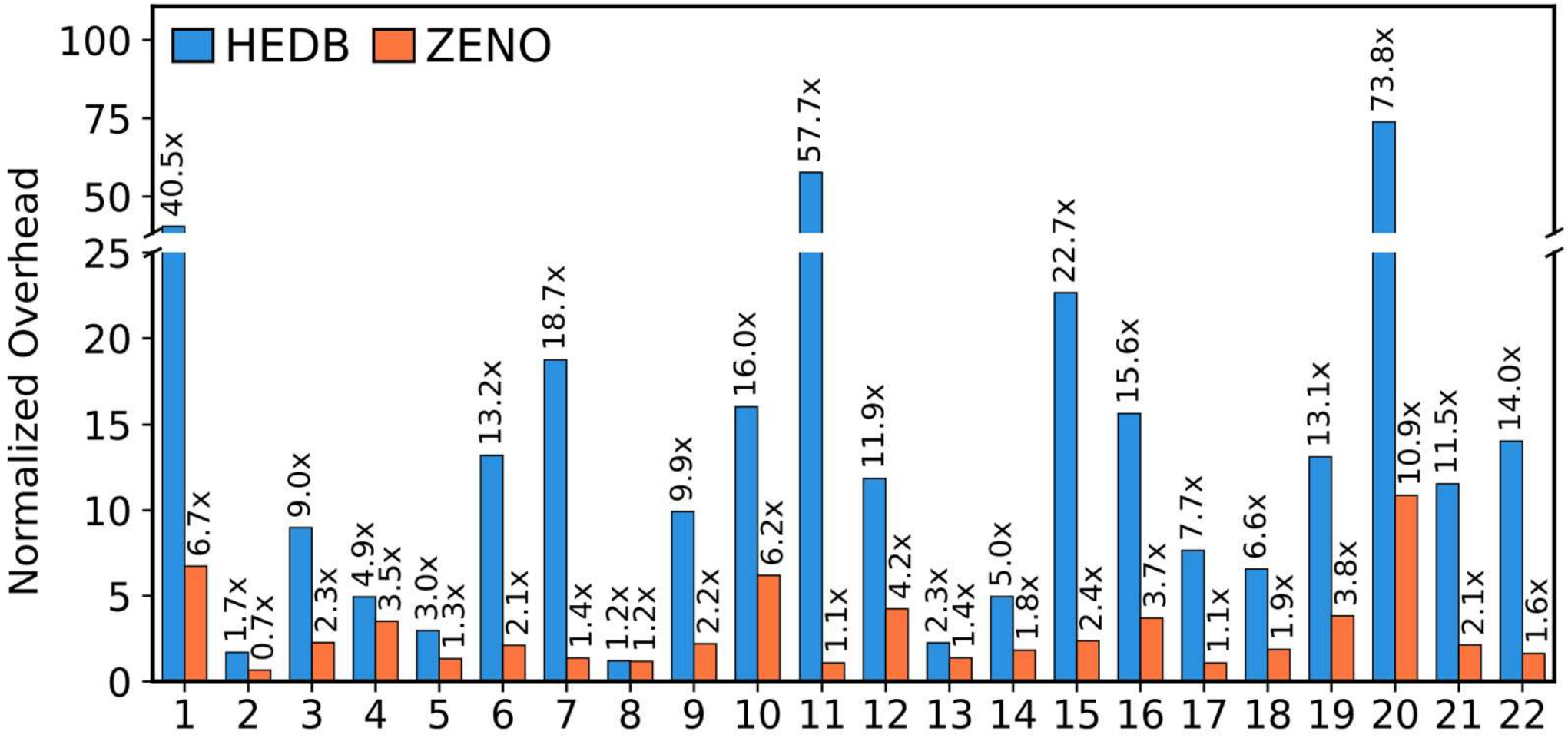
- ***Space reclamation and reuse***
  - An offset array and a slab-like structure for variable-length field management
  - **Temporal partitions** for coordinated value reclamation
- ***Preventing other inconsistencies***
  - **Transaction aborts**: an **MVCC-like approach** retains old mapping versions
  - **System crashes**: coordinated log replay reconstructs mappings
- ***Equivalent* security guarantees**
  - **Plaintext-independence** of FIDs
  - **Leakage analysis**, including positional information and allocation-order leakage in ZENO
- ***Read our paper for more details***

Evaluation setup for ZENO

- **Baselines**:
  - Vanilla PostgreSQL (plaintext)
  - HEDB (a state-of-the-art modern CDB)
- **Workloads**:
  - TPC-H (OLAP) & TPC-C (OLTP)
  - 9 proprietary industrial queries; Sysbench OLTP workload
- **Hardware Environments** (TEEs):
  - ARM S-EL2 (HiSilicon Kunpeng-920, S-EL2 support, 32 vCPUs, 64 GB RAM)
  - Intel x86 TDX (Xeon Platinum, TD Partitioning enabled, 32 vCPUs, 64 GB RAM)

OLAP performance: TPC-H

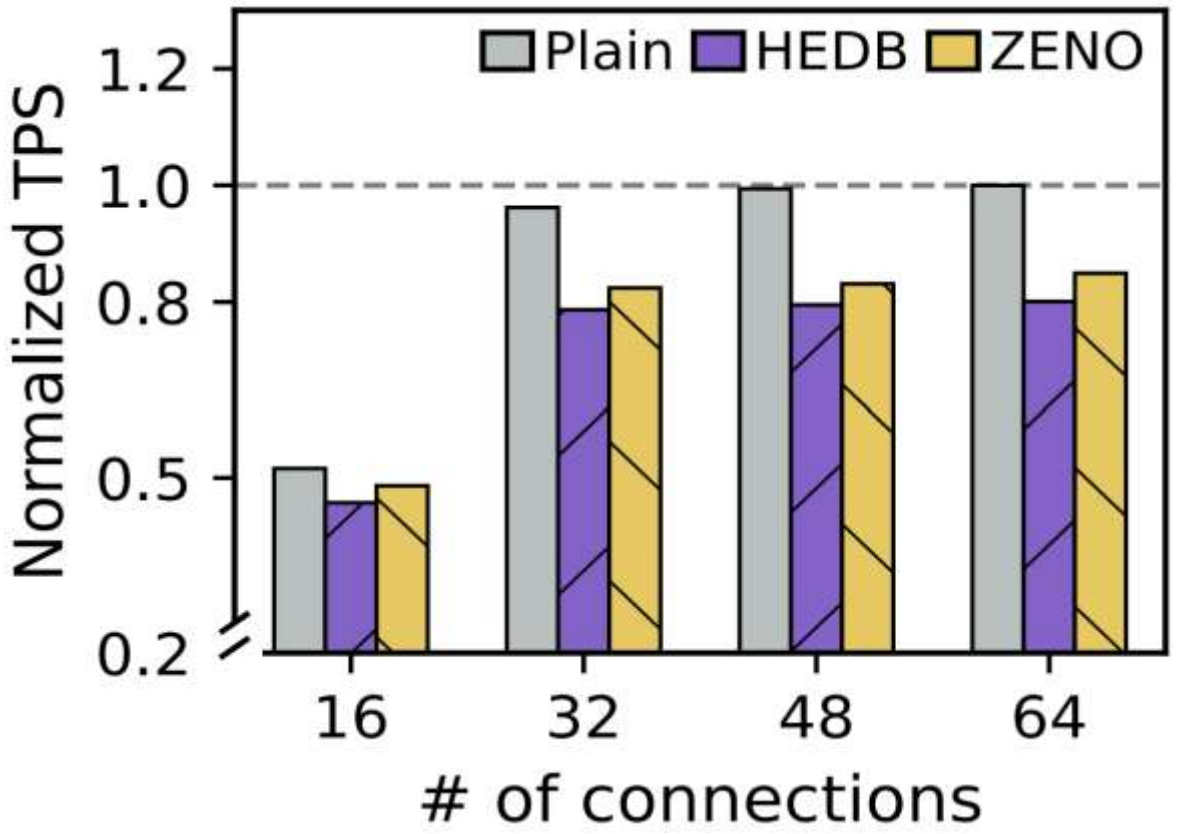
Up to **53.1×** speedup over HEDB



OLTP performance: TPC-C

Reduces HEDB's throughput loss by up to **49.8%**

** Normalized to peak plaintext throughput — **higher is better***



Storage consumption

Reduces storage overhead by **52.8%** on TPC-H and **38.9%** on TPC-C.

	TPC-H	TPC-C
HEDB	3.4×	2.1×
ZENO	1.6×	1.3×

Conclusion

ZENO is a novel CDB design that **decouples indirection from protection** via **crypto-free mappings**.

- Outperforms the SOTA (HEDB) by up to **53.1**× on TPC-H
- Key techniques have been integrated into **GaussDB**
- Open-sourced: **https://github.com/ISCAS-OSLab/ZENO**
- Contact: **huangwx@ios.ac.cn**









