

SharkSQL

the database project

International OpenVMS Forum
Eindhoven, March 7th 2023



Dr. Wolfgang Burger



+43 664 8379528

wolfgang.burger@wdb-tech.com

WHOAMI



- Wolfgang Burger
 - PhD in physics
 - 30 years of experiences in the IT industry
 - Focussed on, but not limited to, OpenVMS, DB management and application programming
 - Until 2019 HPE employee; Master of Technology
- 2019 Founding of the start-up company WDB Tech
 - Goal: SharkSQL development
 - SQL standard compliant distributed relational DBMS system
 - Easy-to-use with all features one would expect from an enterprise level DBMS
 - Enhanced security and data protection features
 - Multi-Platform support – OpenVMS, Windows, Linux (future)
 - Performance, Performance & Performance ...

Topics



- SQL compliancy
 - The ACID principle; facts, rumours and fake statements
- Usability
 - What makes a DBMS easy to use – my thoughts
- Cross DB and distributed transactions
 - SharkSQL GRID access – simple and easy to use
- Security and data protection
 - SharkSQL's enhanced features
- Performance
 - Some benchmark numbers ...
- SharkSQL Connectors/Interfaces

SQL compliancy



- Transactions are a core concept of relational DBMS
- Transactions must adhere to the ACID-principle
 - **A** – Atomic
 - “All-or-nothing” principle; either all or not changes are persistently stored at the end of a transaction.
 - **C** – Consistency
 - A transaction converts the database from one consistent state to another consistent state.
 - **I** – Isolation
 - Isolation determines how restrictively data processed by a transaction are isolated from being accessed by other concurrently executing transaction – we will have a closer look on this topic ...
 - **D** – Durability
 - Durability means that committed data remains permanently stored regardless of whether the RDBMS software is running or not, the server gets rebooted, or in case of any other event or failure.

ACID - fake statements and rumours



“A Lost Update may occur if transactions are executed with isolation level READ COMMITTED”

Original statement (German):

„Das Phänomen *Lost Update*² kann auftreten, wenn das Isolationslevel READ COMMITTED verlangt wird“

[Burkhardt Renz, Professor für Informatik, TU Mittelhessen, Fachbereich MNI, Vorlesungskript “IsolationsLevel in SQL”, SS 2022]

Widespread assumption:

- *The isolation level SERIALIZABLE guarantees that the data is consistent under all circumstances; in other words, with the SERIALIZABLE isolation level you are on the safe side as far as data consistency is concerned.*
- *From the performance perspective optimistic concurrency control is the superior concept (compared with pessimistic concurrency control)*



SQL Standard – Isolation levels

- 1) *P1* (“Dirty read”): SQL-transaction *T1* modifies a row. SQL-transaction *T2* then reads that row before *T1* performs a COMMIT. If *T1* then performs a ROLLBACK, *T2* will have read a row that was never committed and that may thus be considered to have never existed.
- 2) *P2* (“Non-repeatable read”): SQL-transaction *T1* reads a row. SQL-transaction *T2* then modifies or deletes that row and performs a COMMIT. If *T1* then attempts to reread the row, it may receive the modified value or discover that the row has been deleted.
- 3) *P3* (“Phantom”): SQL-transaction *T1* reads the set of rows *N* that satisfy some <search condition>. SQL-transaction *T2* then executes SQL-statements that generate one or more rows that satisfy the <search condition> used by SQL-transaction *T1*. If SQL-transaction *T1* then repeats the initial read with the same <search condition>, it obtains a different collection of rows.

The four isolation levels guarantee that each SQL-transaction will be executed completely or not at all, and that no updates will be lost. The isolation levels are different with respect to phenomena *P1*, *P2*, and *P3*. Table 8, “SQL-transaction isolation levels and the three phenomena” specifies the phenomena that are possible and not possible for a given isolation level.

Table 8 — SQL-transaction isolation levels and the three phenomena

Level	<i>P1</i>	<i>P2</i>	<i>P3</i>
READ UNCOMMITTED	Possible	Possible	Possible
READ COMMITTED	Not Possible	Possible	Possible
REPEATABLE READ	Not Possible	Not Possible	Possible
SERIALIZABLE	Not Possible	Not Possible	Not Possible

SQL Standard – Isolation levels



Isolation level	Lost updates	Dirty reads	Non-repeatable reads	Phantoms
Read Uncommitted	don't occur	may occur	may occur	may occur
Read Committed	don't occur	don't occur	may occur	may occur
Repeatable Read	don't occur	don't occur	don't occur	may occur
Serializable	don't occur	don't occur	don't occur	don't occur



Lost Update – some examples

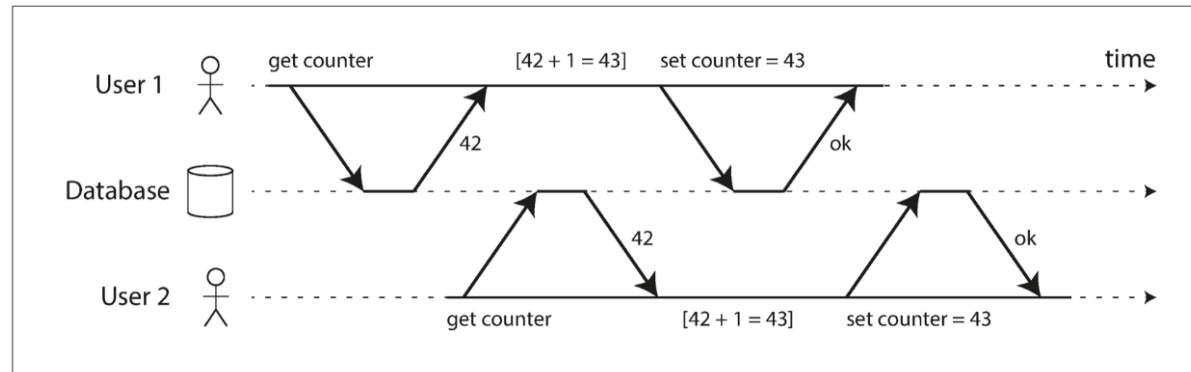


Tabelle 1: Beispiel für Dirty Write

T_1	T_2	Saldo für Konto 1
		100
update Konto set Saldo = 200 where KtoNr = 1		200
	update Konto set Saldo = 250 where KtoNr = 1	250
commit		200
	commit	250

Tabelle 7: Beispiel für Lost Update

T_1	T_2	Saldo für Konto 1
		100
select Saldo from Konto where KtoNr = 1 T_1 liest den Wert 100		
	select Saldo from Konto where KtoNr = 1 T_2 liest den Wert 100	
update Konto set Saldo = 100+100 where KtoNr = 1 ... commit		200
	update Konto set Saldo = 100+50 where KtoNr = 1 ... commit	150



DBMS & Lost Update Prevention

	SQL Standard	Oracle	Oracle/RDB	SQL Server	PostgreSQL	MySQL/MariaDB	SharkSQL
Read Uncommitted							
Read Committed							
Repeatable Read							
Serializable							

Most RDBMS show the Lost Update phenomenon with isolation level READ COMMITTED *but not because it is a consequence of the SQL standard definitions*, but because these DBMS have no mechanism implemented to prevent lost updates.

SERIALIZEABLE guarantees data consistency



SQL Standard

*“The execution of concurrent SQL-transactions at isolation level SERIALIZABLE is guaranteed to be serializable. A serializable execution is defined to be an execution of the operations of concurrently executing SQL-transactions that produces the same effect **as some serial execution** of those same SQL-transactions. A serial execution is one in which each SQL-transaction executes to completion before the next SQL-transaction begins.”*

SERIALIZABLE guarantees data consistency



Simple example; initial values of {X=10, Y=20}

	T1	T2
0	set transaction isolation level serializable;	set transaction isolation level serializable;
1	fetch X	
2		update Y = 50
3		commit;
4	fetch Y	
...	<Any other statements>	
6	commit	

Due to the SQL Standard T1 may fetch either:

- {X=10, Y=20}
- {X=10, Y=50}

From the application's perspective the only valid result set of T1 should be {X=10, Y= 50} since T2 updated Y and committed the update before T1 has fetched Y. Otherwise the application would process outdated data.

SERIALIZABLE guarantees data consistency



- It depends on whether *strict serialization* is supported
 - Pessimistic concurrency control
 - Record/Page locking; implicit time ordering
 - Strict Serialization (i.e. Oracle/RDB)
 - Optimistic concurrency control
 - Basic assumption = no concurrency conflicts; (almost) no locking
 - Concurrency conflicts are typically check during the COMMIT phase
 - If any conflicts are detected the transaction fails and it is rolled back
 - In principle strict serialization is also possible with optimistic concurrency control; requires some sort of timestamp/record version maintenance; if the isolation level SERIALIZABLE is implemented based on transaction snapshotting, strict serialization is usually not supported.
 - ***My personal advice: Read the documentation of the DBMS you are using very carefully and thoroughly.***



Optimistic Concurrency Control - the superior concept?



- Optimistic concurrency control
 - Ongoing transaction:
 - (almost) no locking; no (or few) deadlocks; parallel execution
 - Commit phase:
 - Concurrency issue check; requires to some extent (depends on the isolation level) serialization among the transactions
 - Concurrency issue detected
 - Transaction fails
 - Transaction has to be restarted by the caller
- Pessimistic concurrency control
 - Ongoing transaction:
 - All resources are locked in the mode required by the SQL statement; blocking conditions and deadlocks are very likely when concurrency contention is high; this limits parallel execution
 - Implicit chronological statement ordering
 - Commit phase:
 - Transaction will not fail due to commit to concurrency issues; no additional logic

Optimistic Concurrency Control - the superior concept?



- It depends on the concurrency contention:
 - No or low concurrency contention 
 - High concurrency contention 
- “Real world” OLTP scenario:
 - Some data areas shows high concurrency contention; some others have almost no concurrency contention
- A DBMS typically provides either of the concurrency control models. Examples:
 - Optimistic: PostgreSQL
 - Pessimistic: Oracle/RDB, MySQL/MariaDB
- Only few DBMS support both (solidDB, SQL Server)
- ***SharkSQL supports both Concurrency Control modes***
 - Table property; tables with different concurrency control modes may co-exist within a schema/database; can be dynamically re-defined

Usability – what makes a DBMS easy to use



- The hurdles to working with a DBMS must be as low as possible, even for newcomers. This includes:
 1. DBMS must provide out-of-the-box reasonable performance
 - no expert know-how required to adjust system settings (mostly memory settings)
 - Only basic SQL know-how must be sufficient to create and configure DB objects (i.e. CREATE DATABASE)
 2. Intuitively structured and consistent CLI
 - Consistent method to navigate through all DB objects (i.e. Oracle/RDB)
 - No offline utilities required to manage the DBMS environment
 - Startup/shutdown the environment
 - Backup/restore
 - Task scheduling
 - ...

Easy configuration – an example



Oracle/RDB

```
create database filename disk$db3:[rdb.mydb]mydb
  number of cluster nodes 1
  number of users 1024
  buffer size is 32 blocks
  number of buffer 250
  global buffers are enabled
    (number is 1000000, user limit 1024, page transfer via memory, large memory is enabled)
  create storage area pgbench
    allocation is 65536 pages
    extent is 32769 pages
    page format is uniform
    page size is 16 blocks
    filename disk$db3:[rdb.mydb]mydb_pgbench
    locking is row level;
disconnect all;
alter database filename disk$db3:[rdb.mydb]mydb
  shared memory is system
  notify is enabled
  row cache is enabled
  snapshot is enabled
  journal is enabled (fast commit is enabled)
  add journal mydb_ajj1 filename disk$db3:[rdb.mydb]mydb_ajj1
    allocation is 204800 blocks
    extent is 204800 blocks;
attach 'file DISK$DB3:[RDB.MYDB]MYDB.RDB';
create table accounts (
  aid integer primary key,
  bid integer,
  abalance integer,
  filler character(84)
);
create storage map map_accounts for ACCOUNTS store in PGBENCH disable compression;
create unique index aid on ACCOUNTS (aid asc) type is sorted disable compression store in PGBENCH;
```

SharkSQL

```
create database mydb default directory disk$db3:[sharksql.mydb];
create table accounts (
  aid integer primary key,
  bid integer,
  abalance integer,
  filler varchar(84)
) allocation size 100;
```


Intuitive CLI

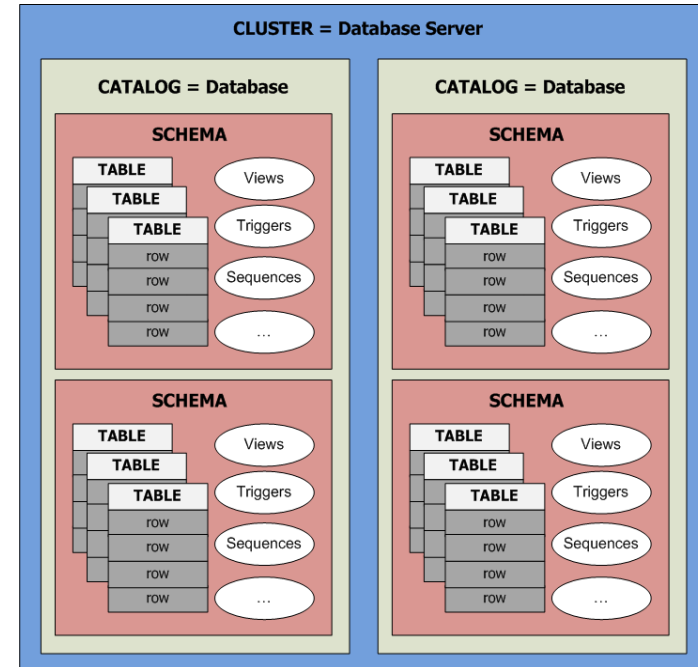


SharkSQL:

```
SHOW DATABASES;  
SHOW DATABASE <db-name>;  
SHOW SCHEMAS;  
SHOW SCHEMA <schema-name>;  
SHOW TABLES;  
SHOW TABLE <table-name>;  
SHOW INDEX ... ON <table-name>;  
...
```

MariaDB:

```
SHOW DATABASES;  
SHOW SCHEMAS;  
SELECT * FROM INFORMATION_SCHEMA.SCHEMATA WHERE SCHEMA_NAME like '<schema-name>;'  
SHOW TABLES;  
SELECT * FROM INFORMATION_SCHEMA.TABLES WHERE TABLE_NAME like '<table-name>;'  
SELECT * FROM INFORMATION_SCHEMA.COLUMNS WHERE TABLE_NAME like '<table-name>;'  
SELECT * FROM INFORMATION_SCHEMA.TABLES_CONSTRAINTS WHERE TABLE_NAME like '<table-name>;'  
SELECT * FROM INFORMATION_SCHEMA.STATISTICS WHERE TABLE_NAME = '<table-name>;'  
SELECT * FROM INFORMATION_SCHEMA.PARTITIONS WHERE TABLE_NAME like '<table-name>;'  
...
```





SharkSQL SHOW command

Principles:

- Full information about the DB object requested
- Brief information about all associated and/or child DB objects

```
SharkSQL> show database mydb;

SHARKSQL database: MYDB
-----
Access Host: LocalHost
Description: ---
Create version: V5.0
Root File: C:\MYDB\MYDB.DBR
Default Directory: C:\MYDB
Default Character Set: UNSPECIFIED
Collate: BINARY
Owner: ADMIN
World Privileges: ()
AIJ Directory: C:\MYDB
enabled: Yes
Initial/Extend Size: 256 [MB]
Default Snapshot Directory: C:\MYDB
Default Backup Directory: SHARKSQL$BCK
Connection limit: 16384
Current connections: 1
Open mode: automatic
Node Limit: unlimited
Current Backup Index: 0
Last valid Backup Index: 0
Full Backup required: Yes

Schemas in database: MYDB
-----
INFORMATION_SCHEMA Information schema
MYDB MYDB DB default schema

OK, 0 row processed, (0.000 sec).
```

```
VMSTM2
SharkSQL> show table accounts;

Table 'ACCOUNTS' in schema 'PGBENCH.PGBENCH':
-----
Description: ---
Revision: 1
Default Character Set: UNSPECIFIED
Collate: BINARY
AutoCommit: No
Auto-Increment: No
Caching enabled: Yes
Concurrency control: Pessimistic
Row Compression: Yes
Low Limit: 0 [%]
Data Encryption: No
Table Space size: 868 [MB]
Table Space free: 47 [MB]
Min. Partition Size: 100 [MB]
Partition Extend Size: 100 [MB]
Page Size: 8 [kB]
LOB Chunk Size: 32 [kB]
Store clause: PGBENCH
Section clause: ---
Partition clause: ---
Number of Sections: 1
Partitions/Section: 1
Compute Signature: No

Column Name          Data Type          Description
-----
aid                  int32
bid                  int32
abalance             int32
filler               varchar(84)
CHARSET UNSPECIFIED COLLATE BINARY

Column Name          CharSet            Collation
-----
filler               UNSPECIFIED        BINARY

'ACCOUNTS' indices
-----
Index                Index Type         Description
-----
AID                  B-TREE|PRIMKEY    (none)
BID                  B-TREE|DUP|LCL    (none)

'ACCOUNTS' constraints
-----
Name                Constraints
-----
AID                  PRIMARY KEY (aid)
DEFFERRABLE INITIALLY IMMEDIATE

'ACCOUNTS' storage map
-----
Storage Area         Section            Encrypted          Constraint         Size/Free [MB]
-----
PGBENCH              0                  No                ---                868/47
{0}

OK, 0 row processed, (0.050 sec).
```

Distributed & Cross-Database transactions



- A distributed transaction operates on data that is located in 2 or more databases managed on different nodes
- If the databases are located on the same node, such transactions are called Cross-Database transactions
 - Side note: PostgreSQL supports distributed but no cross-database transactions
- Nothing new; other DBMS like Oracle and PostgreSQL support distributed transaction
- Vehicle to access remote data are so called federations or database links
 - DB links have to be configured manually
 - From a client point of view a federation is a static method
 - Access to linked remote databases is always limited to some extent. Typically, only DML statements can be executed.
 - Particular attention has to be paid to security when a DB link is configured. DB links bear the risk of piggy-back security holes.

Configuration example PostgreSQL: DB-Link via Foreign Data Wrapper



1) Create required objects on OMEGA to access remote tables

- SALES.PAYMENTS on ALPHA001
- SALES.ACCOUNTS on ALPHA002

```
CREATE SERVER alpha FOREIGN DATA WRAPPER postgres_fdw OPTIONS (host 'alpha001', dname 'sales')
CREATE SERVER beta FOREIGN DATA WRAPPER postgres_fdw OPTIONS (host 'alpha002', dname 'sales')
CREATE USER MAPPING FOR this_user SERVER alpha (user 'postgres', password 'fdw_password');
CREATE USER MAPPING FOR this_user SERVER beta (user 'postgres', password 'fdw_password');
IMPORT FOREIGN SCHEMA public LIMIT TO (payments) FROM SERVER alpha INTO public;
IMPORT FOREIGN SCHEMA public LIMIT TO (accounts) FROM SERVER beta INTO public;
```

Client



OMEGA

ALPHA001

ALPHA002

2) Now we can execute Query:

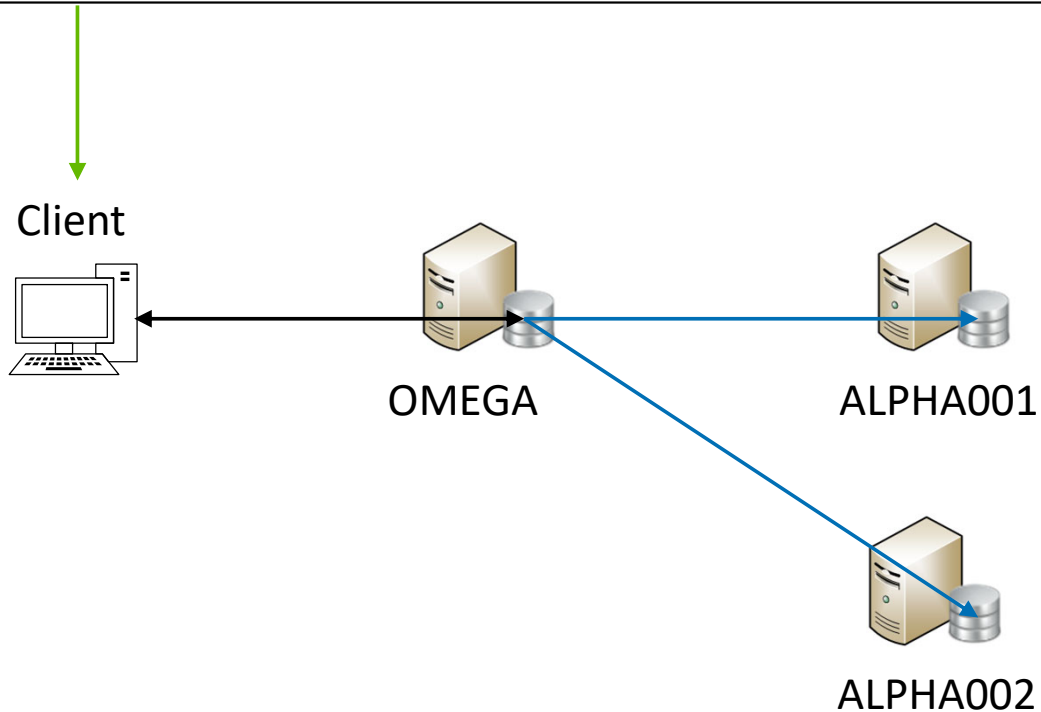
```
SELECT pm.type, acct.balance FROM accounts acct
LEFT JOIN payment_methods pm ON pm.act_id = acct.id
WHERE acct.balance > 0;
```



PostgreSQL: Using dblink

We can execute the query immediately:

```
SELECT pm.type, acct.balance FROM
    dblink ('host=alpha001 user=postgres password=fwd_password dbname=sales', 'select balance, id from sales where balance > 0')
    AS acct(balance integer, id integer)
LEFT JOIN
    dblink ('host=alpha002 user=postgres password=fwd_password dbname=sales', 'select type, act_id from sales')
    AS pm(type varchar(255), id integer)
ON pm.act_id = acct.id;
```

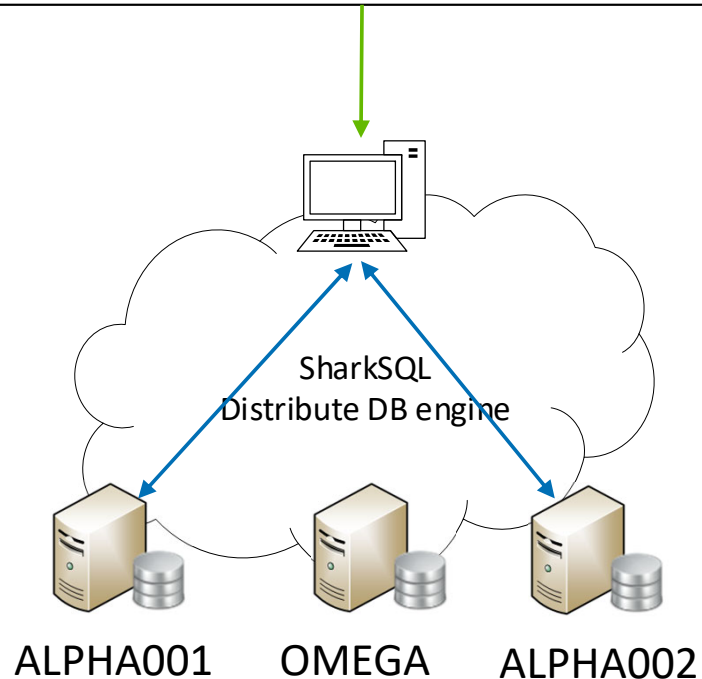


SharkSQL GRID access



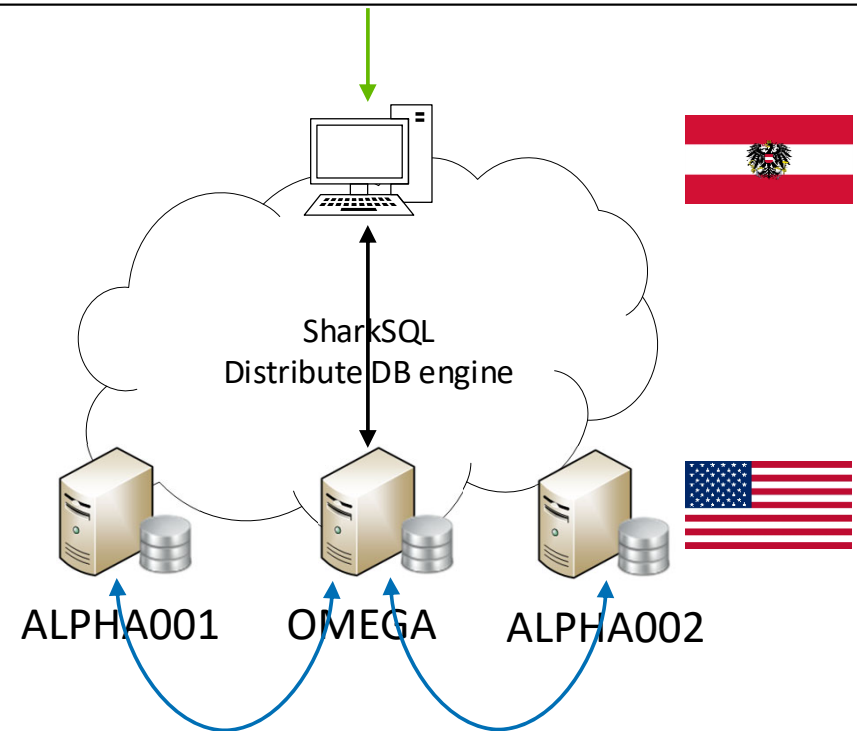
Direct connection:

```
CONNECT TO sales@alpha001 USER system PASSWORD <password>;  
CONNECT TO sales@alpha002 USER system PASSWORD <password>;  
SELECT pm.type, acct.balance FROM alpha001.sales.accounts acct  
LEFT JOIN alpha002.sales.payment_methods pm ON pm.act_id = acct.id  
WHERE acct.balance > 0;
```



Connect via OMEGA:

```
SET DEFAULT CONNECTION omega;  
CONNECT TO sales@alpha001 [ USER system PASSWORD <password> ];  
CONNECT TO sales@alpha002 [ USER system PASSWORD <password> ];  
SELECT pm.type, acct.balance FROM alpha001.sales.accounts acct  
LEFT JOIN alpha002.sales.payment_methods pm ON pm.act_id = acct.id  
WHERE acct.balance > 0;
```



SharkSQL GRID access



- SharkSQL DB engine is a distributed DB engine
 - Remote databases can be accessed directly from the client or via an access server
 - No configuration effort; SQL CONNECT statement only
 - No risk of piggy-back security holes; credential provided by the client are used to login to the remote DBMS
 - No statement limitation; a user can execute all DML and DDL statements on the remote DBMS for which he is privileged.
 - SQL statements remain syntactically strictly following the language definitions of the SQL standard
- ***The Client has always a “local” view on all connected DBs regardless if they are locally attached or remote***

SharkSQL GRID access



- Highly secure (“unhackable”) network encryption
 - Encryption key changes after each network packet
 - Peers calculate the new encryption key independently based on the context of the last packet and the old encryption key value
 - The value of the new encryption key also determines whether the packet content:
 - Will be compressed
 - Content is reordered within the packet
 - The peers independently calculate the initial key using a quantum-mechanical eigenvalue equation; the boundary conditions for this equation and the harmonic overtone selected from the solution set are determined by the content of the client's WHOAMI message; the selected harmonic overtone is the input for key generation.

Performance – Benchmark-Test



- OpenVMS
 - VSI OpenVMS V8.4-2L1 / VSI OpenVMS V8.4-2L3
 - HPE rx2800 i4 (Intel Itanium 9540 8-core/2.13GHz/24.0MB)
 - Storage: EVA4 4400
- Windows
 - Windows 10 Pro, Version 21H2, Build 19044.2604
 - HP EliteBook 840 G5, Intel® Core™ i7-8550U CPU @ 1.80GHZ 1.99
 - 16GB RAM
 - 512 GB PCIe® NVMe™ M.2 SSD



Performance – Benchmark-Test

- MariaDB
 - OpenVMS: MariaDB 5.5-63
 - Windows: MariaDB 10.11.2
 - Crash save configuration:
 - innodb_flush_log_at_trx_commit = 1
 - sync_binlog = 1
- PostgreSQL:
 - OpenVMS: PostgreSQL 13.3
 - Windows: PostgreSQL 15.2
 - Memory adjustments:
 - shared_buffers = 8GB (default: 128 MB)
 - work_mem = 64MB (default: 4 MB)

Performance – Benchmark-Test



- Oracle/RDB
 - OpenVMS: Oracle Rdb SQL V7.3-320
- SharkSQL
 - OpenVMS: SharkSQL V5.0 (pre-release)
 - Global cache size: 32 GB (Installation default)
 - Windows: SharkSQL V5.0 (pre-release)
 - Global cache size: 6 GB (Installation default: 8GB)

Performance



- PGBENCH Test (TPC-C like; standard PostgreSQL test)
- Easy to implement and the ultimate "killer" test for concurrency issues:
 - Updating 3 different tables (50, 500, 5.000.000 records)
 - 1 Insert
 - 1 Select

```
\set aid random(1, 100000 * :scale)
\set bid random(1, 1 * :scale)
\set tid random(1, 10 * :scale)
\set delta random(-5000, 5000)
BEGIN;
UPDATE pgbench_accounts SET abalance = abalance + :delta WHERE aid = :aid;
SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
UPDATE pgbench_tellers SET tbalance = tbalance + :delta WHERE tid = :tid;
UPDATE pgbench_branches SET bbalance = bbalance + :delta WHERE bid = :bid;
INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES (:tid, :bid, :aid, :delta, CURRENT_TIMESTAMP);
END;
```



Performance - PGBENCH

- 1. Test, default isolation levels used, OpenVMS and Windows
 - Oracle/RDB: SERIALIZABLE
 - MariaDB: REPEATABLE READ
 - PostgreSQL*: READ COMMITTED
 - SharkSQL: READ COMMITTED

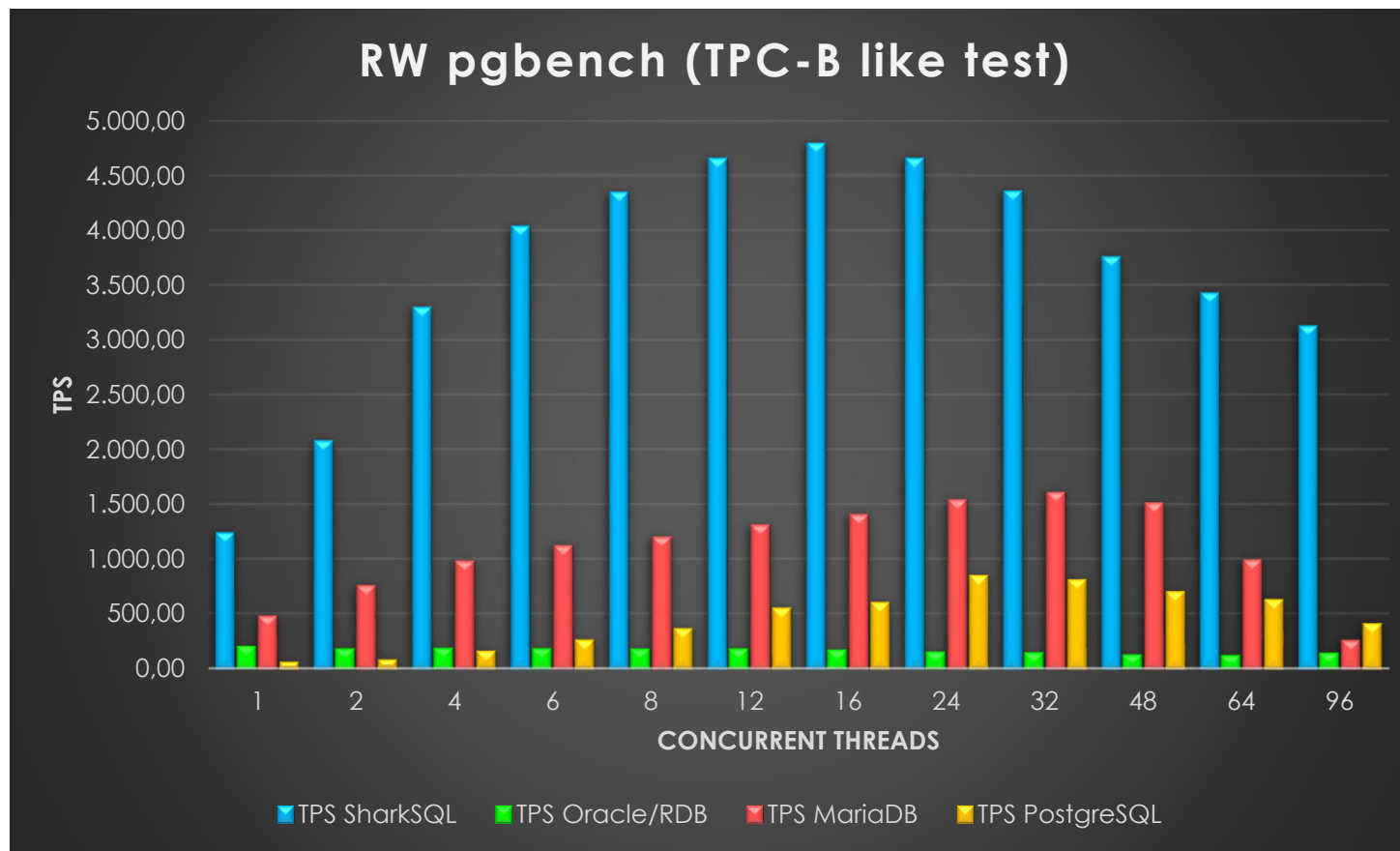
* PostgreSQL PGBENCH test with READ COMMITTED succeeds but the data is corrupted; PostgreSQL isolation level READ COMMITTED does not prevent lost updates; without lost update prevention this test setup results in lost updates.

- 2. Test, using lost update prevention isolation level on Windows
 - MariaDB: REPEATABLE READ
 - PostgreSQL: REPEATABLE READ
 - SharkSQL: READ COMMITTED



PGBENCH – OpenVMS Read-Write results

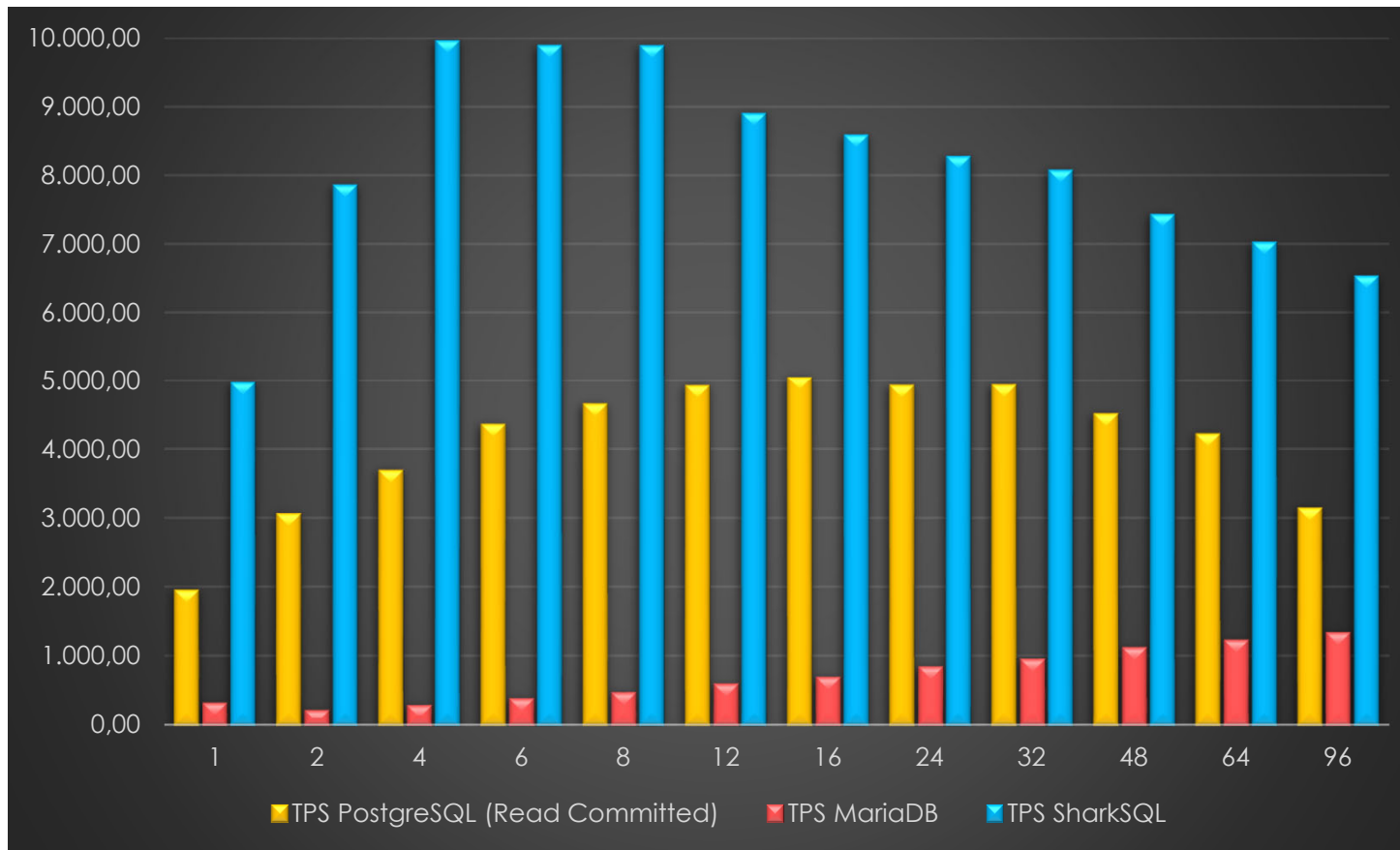
Default isolation levels used for all DBMS tested





PGBENCH – Windows 10 Read-Write results

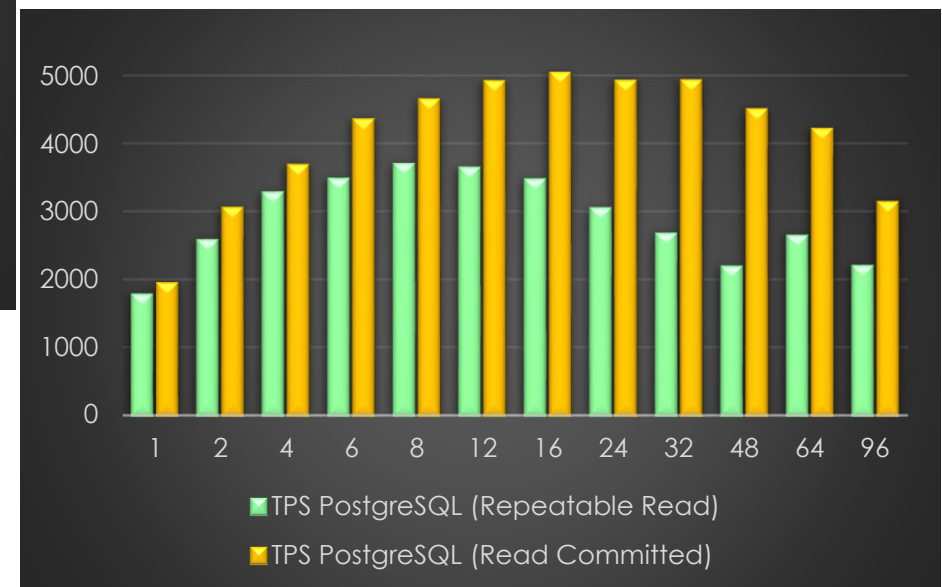
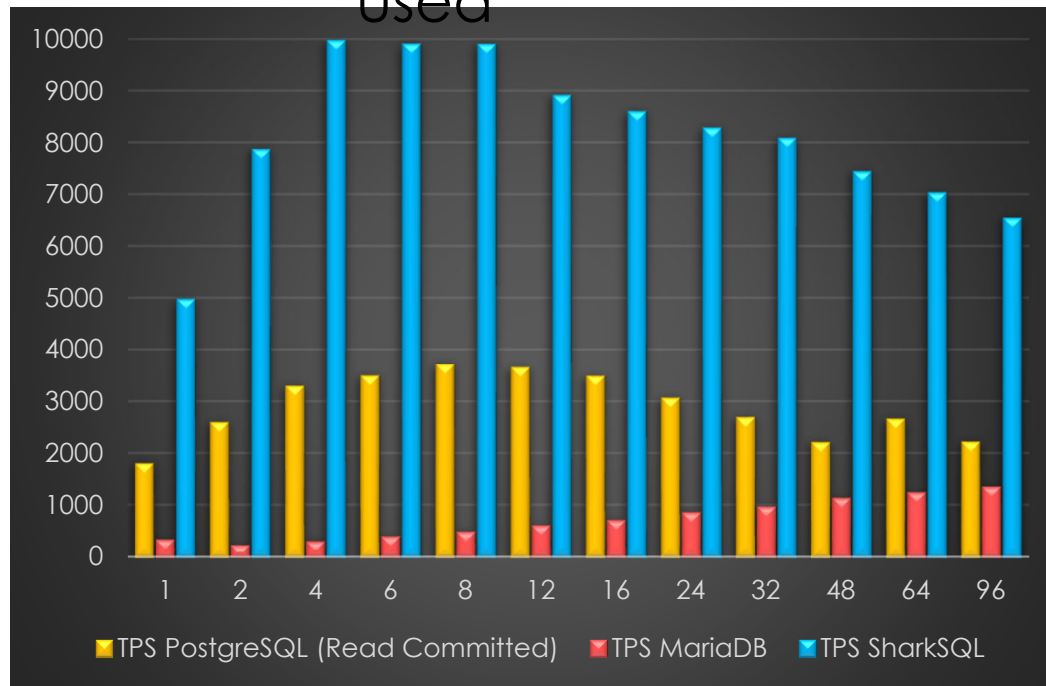
Default isolation levels used for all DBMS tested





PGBENCH – Windows 10 Read-Write results

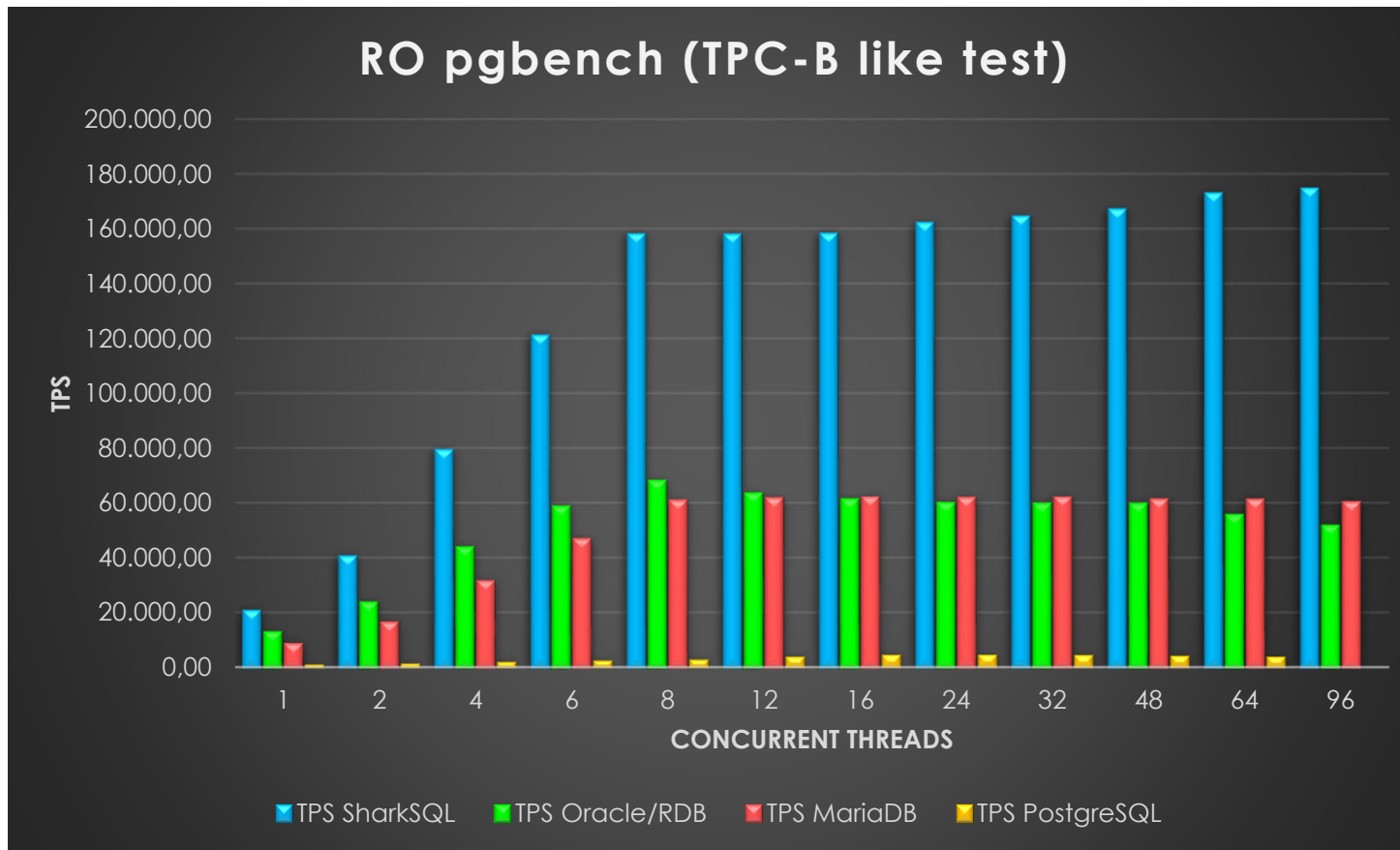
P4 Lost update preventing isolation level used





PGBENCH – OpenVMS Read-Only results

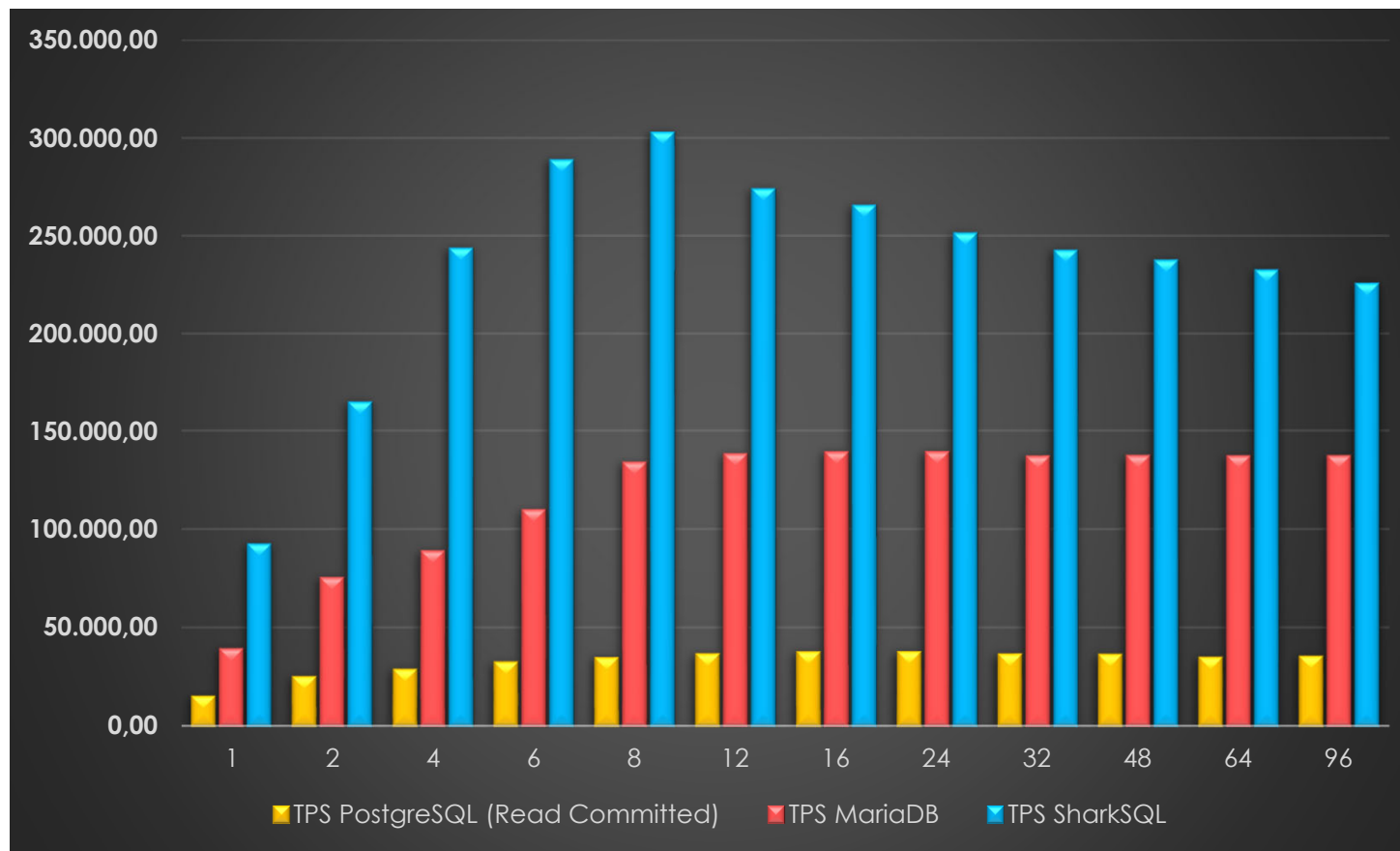
Default isolation levels used for all DBMS tested





PGBENCH – Windows Read-Only results

Default isolation levels used for all DBMS tested



Single Stream – Bulk insert & update



Test table definition:

```
CREATE TABLE test_table  
  (iIndex integer primary key,  
   qTime timestamp,  
   sName character(255));
```

(Oracle/RDB only)

```
CREATE UNIQUE INDEX iIndex on test_table (iIndex asc) TYPE IS SORTED ENABLE COMPRESSION STORE IN test;
```

Insert:

```
TRUNCATE TABLE test_table;  
INSERT INTO test_table select * from load_table;
```

Update (Oracle/RDB, PostgreSQL, SharkSQL):

```
UPDATE test_table  
  set qTime = localtimeStamp,  
      sName = cast(iIndex as char(16)) || '. Entry';  
UPDATE test_table  
  set qTime = localtimeStamp,  
      sName = sName || ' (manually)';
```

Update (MariaDB):

```
UPDATE test_table  
  set qTime = localtimeStamp,  
      sName = concat(cast(iIndex as char(16)), '. Entry');  
UPDATE test_table  
  set qTime = localtimeStamp,  
      sName = concat(sName, ' (manually)');
```

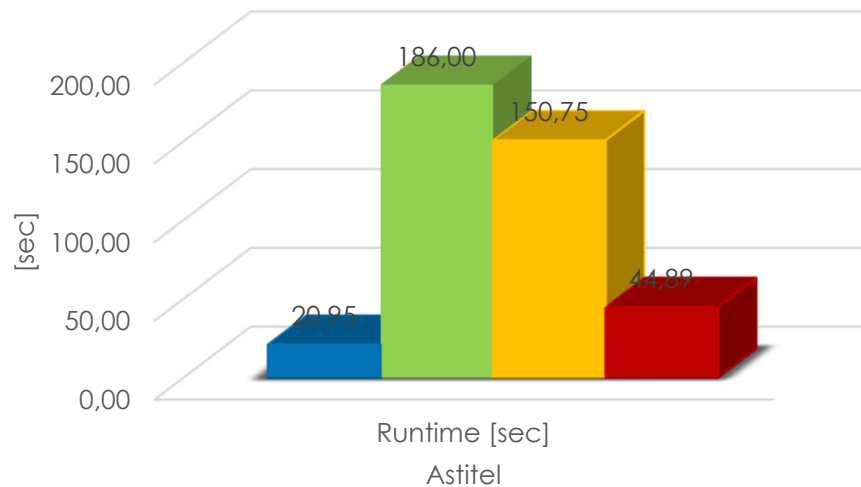
Single Stream – Bulk insert & update



VSI OpenVMS V8.4-2L3

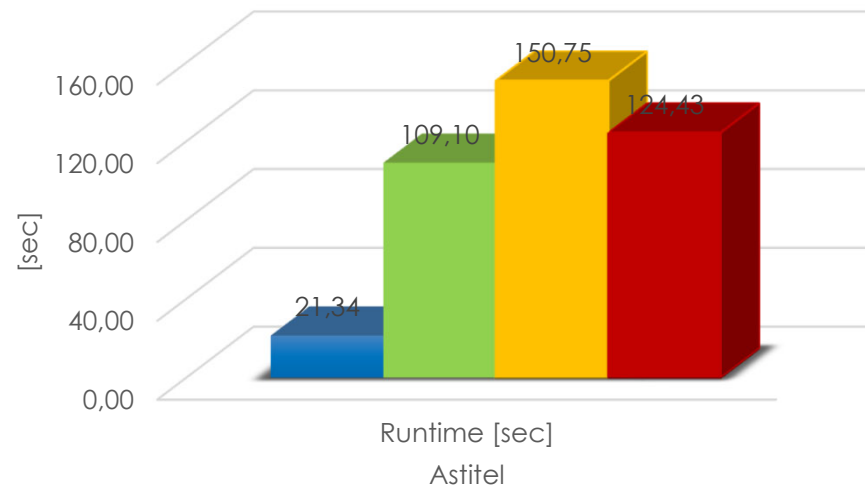
HPE rx2800 i4 (Intel Itanium 9540 8-core/2.13GHz/24.0MB)

Insert 2.000.000 rows



■ SharkSQL ■ Oracle/RDB ■ PostgreSQL ■ MariaDB

Update 2.000.000 rows



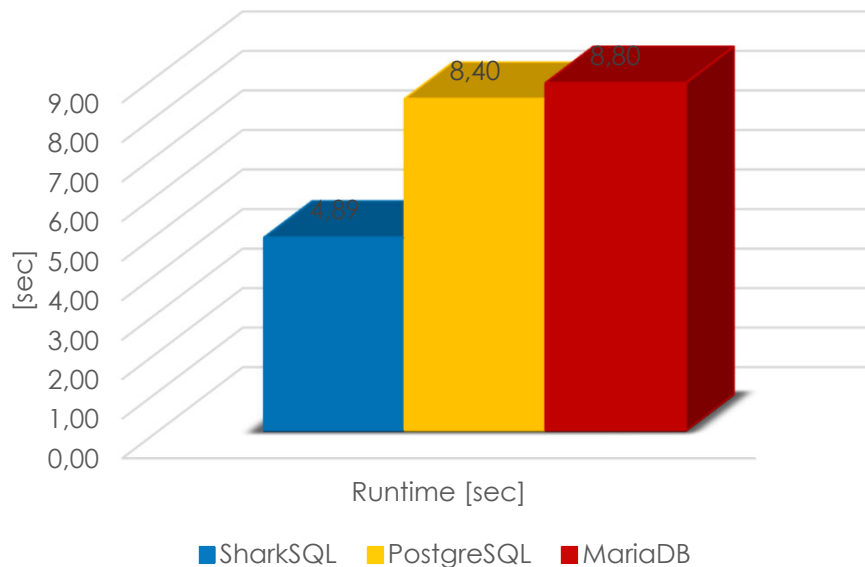
■ SharkSQL ■ Oracle/RDB ■ PostgreSQL ■ MariaDB

Single Stream – Bulk insert & update

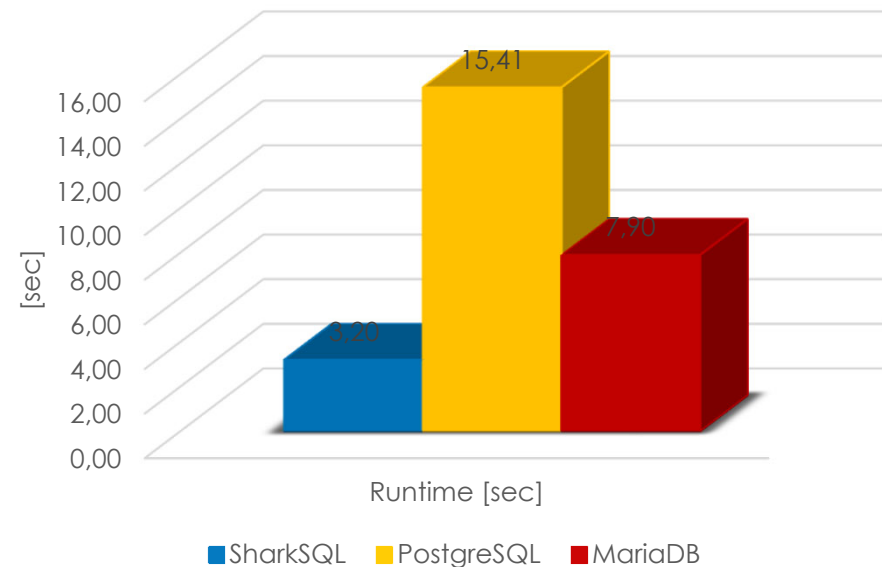


Windows 10 Pro, Version 21H2, Build 19044.2604
HP EliteBook 840 G5, Intel® Core™ i7-8550U CPU @ 1.80GHZ 1.99

Insert 2.000.000 rows



Update 2.000.000 rows



Performance – Grouping and sorting



1. Test

```
SELECT COUNT(distinct iNbr) from test_sort;
```

Table definition (2.000.000 records, 10.000 distinct iNbr values):

```
CREATE TABLE test_sort  
    (iIndex integer primary key,  
     qTime timestamp,  
     sName varchar(255)  
     iNbr bigint);
```

(Oracle/RDB only)

```
CREATE UNIQUE INDEX iIndex on test_sort (iIndex asc) TYPE IS SORTED ENABLE COMPRESSION STORE IN test;
```

2. Test

```
SELECT iNbr, COUNT(sName) from test_sort_2 GROUP BY iNbr HAVING count(sName) > 3;
```

Table definition (2.000.000 records, 1.223.500 distinct iNbr values):

```
CREATE TABLE test_sort_2  
    (iIndex integer primary key,  
     qTime timestamp,  
     sName varchar(255)  
     iNbr bigint);
```

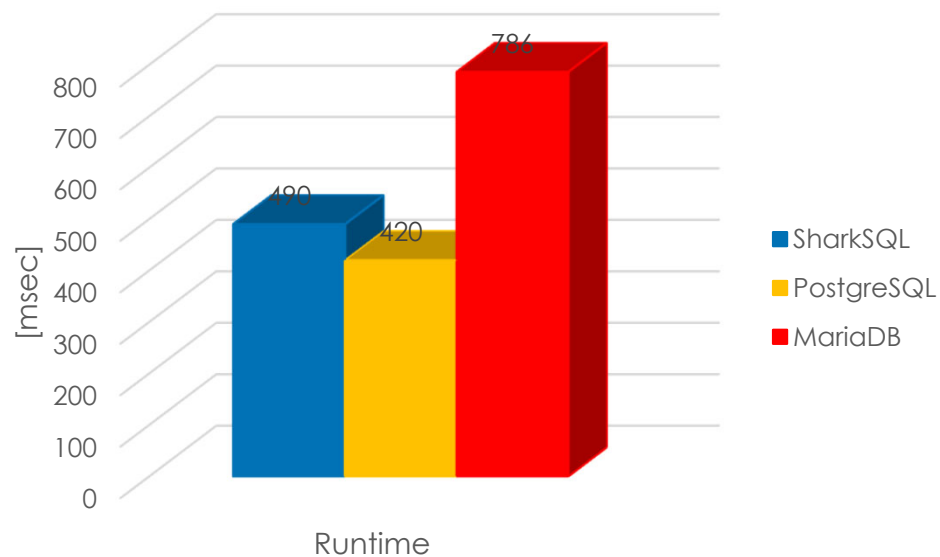
(Oracle/RDB only)

```
CREATE UNIQUE INDEX iIndex on test_sort_2 (iIndex asc) TYPE IS SORTED ENABLE COMPRESSION STORE IN test;
```

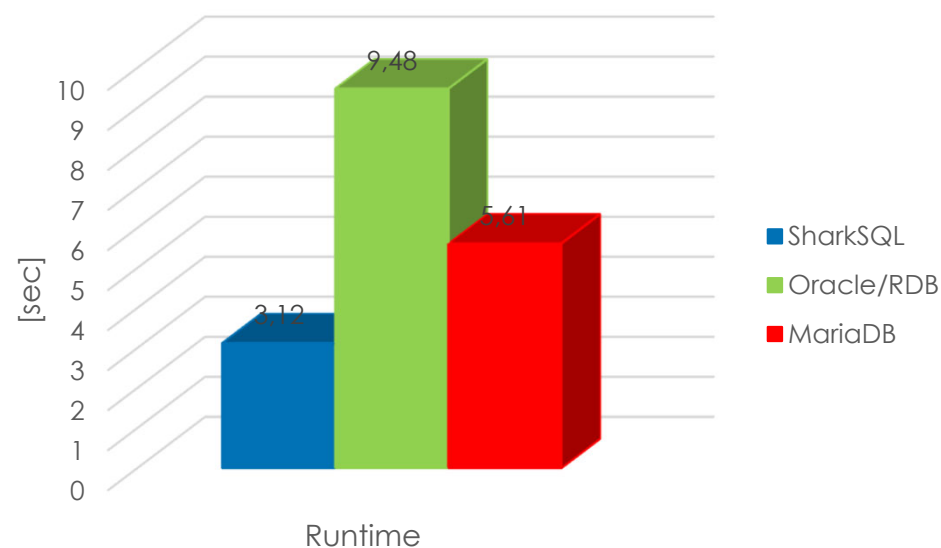


SELECT COUNT(distinct iNbr) from test_sort;

Window 10 Pro



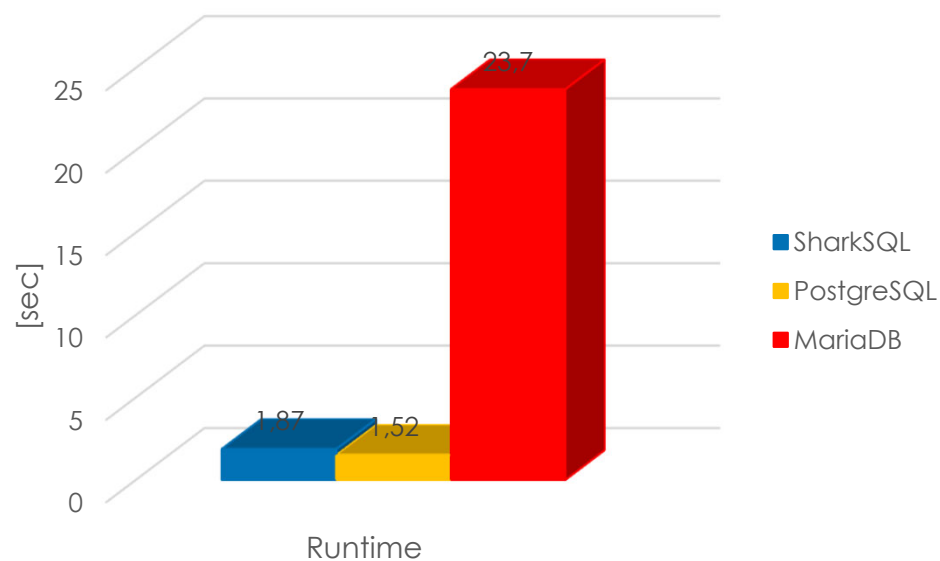
VSI OpenVMS V8.4-2L3



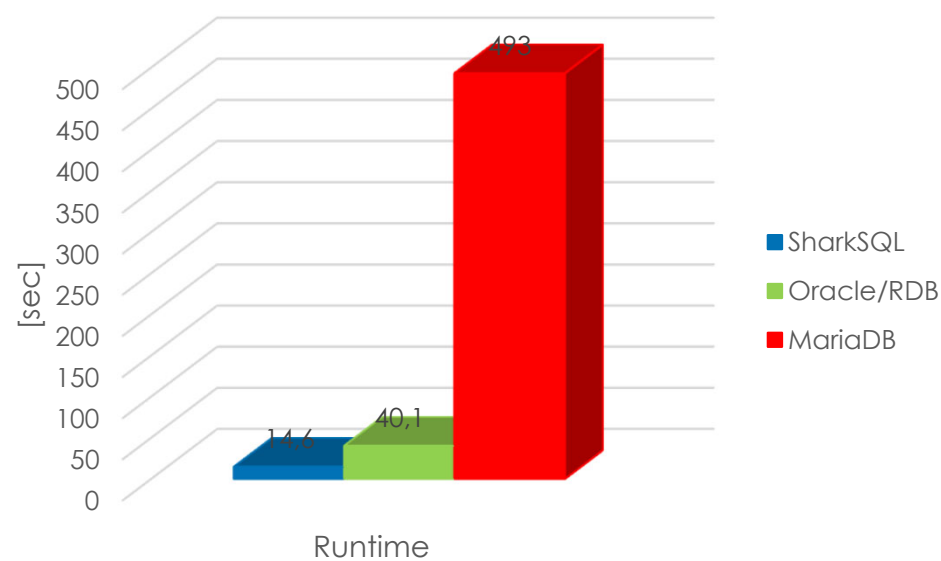


SELECT iNbr, COUNT(sName) from test_sort_2 GROUP BY iNbr ...

Window 10 Pro



VSI OpenVMS V8.4-2L3



SharkSQL Connectors/Interfaces



- Embedded SQL (initial release: C only)
- CVM pre-compiler
 - comparable to Oracle/RDB SQL Module Processor
- SSTR (Secure Service and Transaction Router)
- Full MySQL/MariaDB wire protocol support
 - Any connector available for MySQL/MariaDB can be used to access SharkSQL:
 - JDBC
 - ODBC
 - ADO.NET
 - C++ driver
 - PHP
 - Perl
 - Ruby
 - Go
 - Rust
- Native connectors (especially ODBC and JDBC) will be provided

SharkSQL Roadmap



- Release plan:
 - Field test release mid2023
 - Production release end 2023
- Support Versions
 - OpenVMS IA64
 - VSI OpenVMS V8.4-2L1 and higher
 - HPE OpenVMS V8.4 (can be backup-ported customer on request)
 - OpenVMS x86
 - VSI OpenVMS 9.2 and higher
 - Windows
 - Windows 10/11
 - Windows Server 2016/2019/2022

SharkSQL Roadmap



- Development plan (2024 – 2025)
 - Linux port
 - Database “shadowing”/database replication
 - Adding new SQL features
 - GROUPING SETS, CUBE and ROLLUP
 - Enhanced Window functions (currently only basic features have been implemented).
 - RMS file mapping (OpenVMS only)
 - Native Connectors (ODBC, JDBC etc.)
 - MySQL/MariaDB server integration



Terms & Conditions

- SharkSQL will be distributed as shareware
- Everyone can use it for free
- Customers will be charged only for
 - Support
 - Professional Service (Consulting and Training)
- For more information please contact me directly or send a mail to: sharksql@wdb-tech.com
- Thank you for your attention

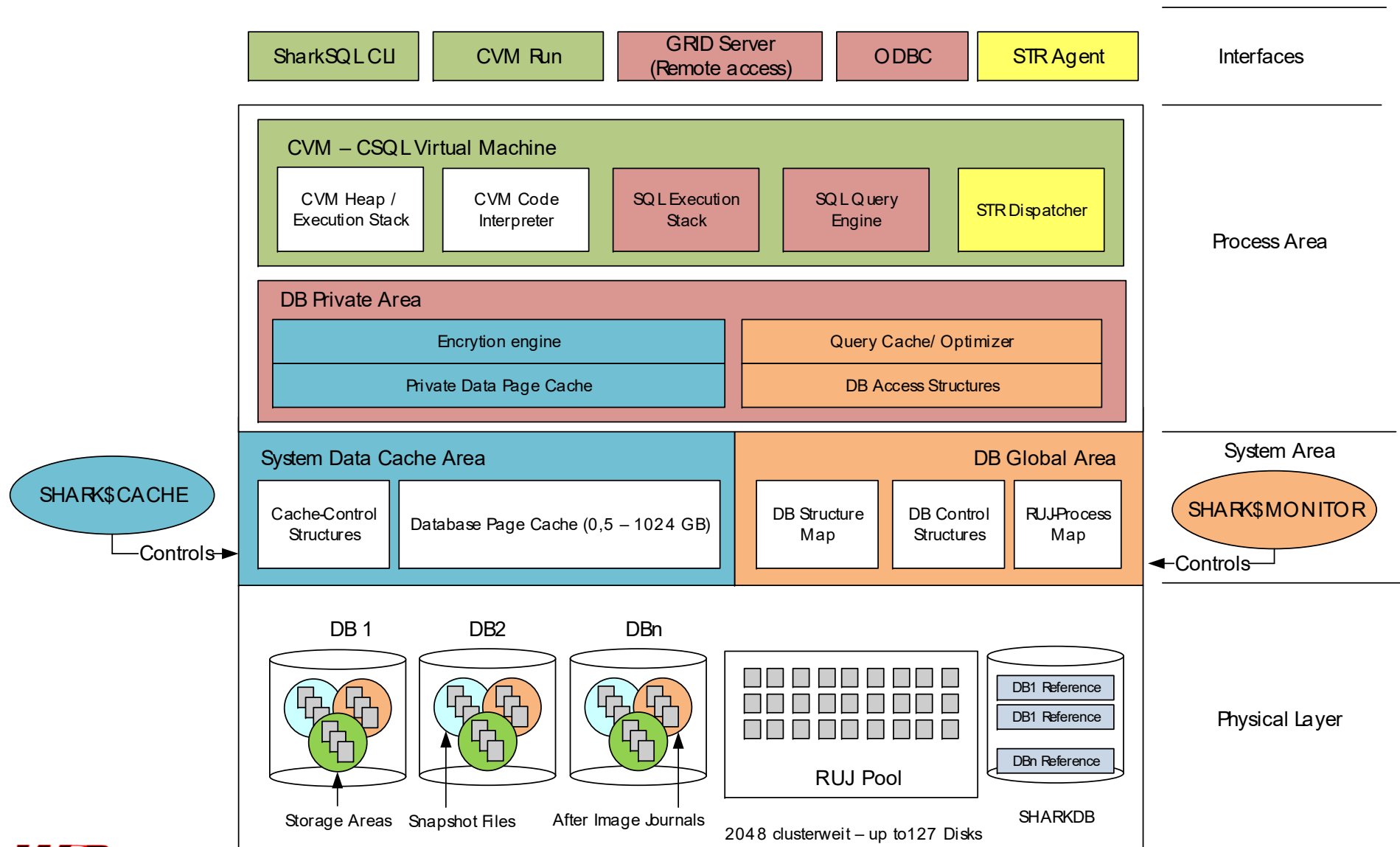
Questions



Backup/Additional Info Slides



Architecture/Components





RUJ user mapping

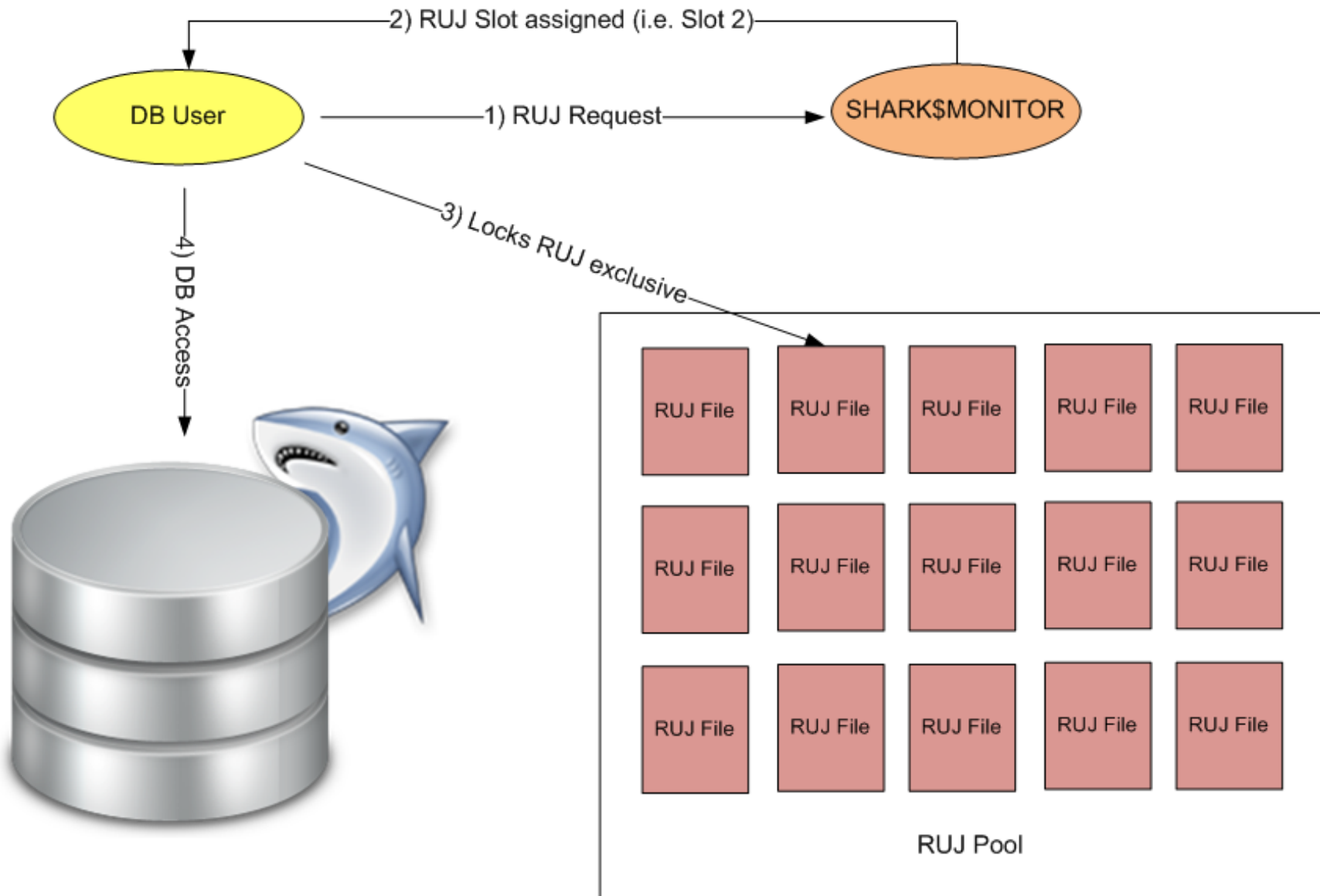
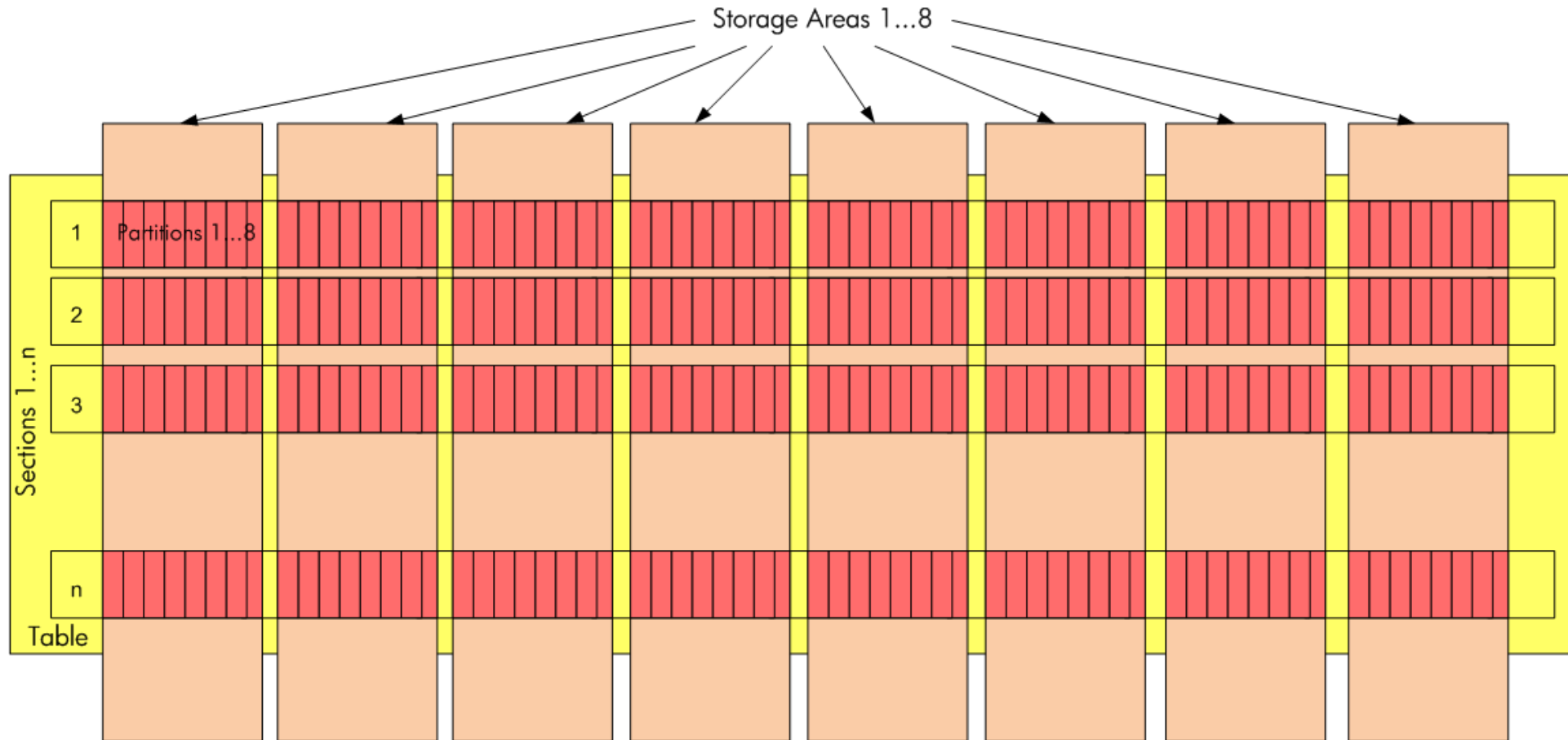


Table Partitioning



Max. Partitions = 8 StgAreas * N Sections * 32 Section-Partitions



Task Scheduler

- Automates periodic tasks
- Up to 64 tasks can be executed in parallel
- Configuration items
 - Cmd to execute:
 - Cmd line scripts
 - Host programs
 - CSQL programs
 - Start time
 - Schedule
 - Yearly
 - Monthly
 - Weekly
 - Daily
 - Free definable time interval
 - Single shot

Task Scheduler



- Schedule exclude list [optional]
 - i.e. schedule DAILY exclude (Sa, So)
 - means – run daily but not on Saturday and Sunday
- Execution node list [optional]:
 - If not defined – task executed on all cluster members
 - 1. node in the list defines the primary execution node
 - 2. – n. node = standby execution node
 - If the task scheduler is running on the primary node it is executed on this node
 - If not, the task is executed on one of the standby nodes defined in order of their appearance in the execution node list.

Backup/Restore



- DB backup
- Schema backup
- Backup options:
 - Online backup:
 - full
 - incremental
 - no data
 - compressed
 - Offline backup
 - full
 - incremental
 - no data
 - compressed
 - physical

Backup/Restore



- Only backup to disk supported
- PURGE clause
 - On success:
 - all AIJs except the AIJ of the current day are automatically purged
 - Locally stored backup sets are purged if the backup is a full backup
 - a backup set consists of a full backup plus all adjacent incremental backups

Backup/Restore



- Restore

- RESTORE DATABASE <DB name>
 - Searches in SHARK\$BCK directory for the most recent valid DB backup set
- RESTORE SCHEMA <Schema name>
 - Searches in SHARK\$BCK directory and the DB backup default directory for the most recent valid SCHEMA backup set
- Automatically restores data from existing AIJs
 - Except the NO AIJ clause is specified with the RESTORE command
- Index build can be suppressed to speed up restore
 - NO INDEX clause must be specified
 - Indices must be manually build with the ALTER TABLE ... BUILD INDEX command



Results - OpenVMS

– Systems:

- ES47 2P/1C 1GHz, 2 GB Memory (2005) HPE OpenVMS V8.4 AXP
- rx4640 1P/1C 1.3 GHz, 2GB Memory (2006) HPE OpenVMS V8.4 I64
- 2x EVA4400 (2011)

– DB writers:

- 3 → 2 on ES47, 1 on rx4640
- 1 GB cluster-wide SharkSQL DB cache

– Overall performance:

- 3600 Terminals

Ticket transactions / sec – long term load (> 30 min)	~ 1200 Tps
Ticket transactions / sec – peak load (30 – 60 sec)	> 1500 Tps
1000 concurrent Terminal/User Login requests	3 - 5 sec
DB-recovery time after crash of a cluster member processing high load (~ 400 Tps)	~ 1 sec



Results - OpenVMS

- System:
 - Rx2800 i4 4P/16C (2016) VSI OpenVMS V8.4-2L1
 - 2x EVA4400 (2011)
- DB writers:
 - 8
- Performance:
 - 1800 Terminals

Ticket transactions / sec – long term load (> 30 min)	~ 2500 Tps
Ticket transactions / sec – peak load (30 – 60 sec)	~ 2500 Tps
1000 concurrent Terminal/User Login requests	< 3 sec
DB-recovery time after crash of a DB writer process (load ~ 400 Tps)	< 1 sec

Note: I/O bound, overall CPU load ~120% - just a bit more than 1 Core