Hyperledger Fabric 1.4.0
Performance Information Report

N K Lincoln
IBM Blockchain Developer Tools
nick.lincoln@uk.ibm.com

Table of Contents

Preface ... 3
Notes ... 3
Benchmark Information ... 4
System Under Test ... 4
The Smart Contract ... 4
Smart Contract Benchmarks ... 6
Empty Contract Benchmark Results ... 8
Evaluate ... 8
Evaluate Results ... 8
Evaluate Observations ... 10
Submit ... 11
Submit Results ... 11
Submit Observations ... 13
Evaluate Transaction Benchmark Results ... 15
Get Asset Benchmark ... 15
Benchmark Results ... 16
Benchmark Observations ... 18
Batch Get Asset Benchmark ... 19
Benchmark Results ... 19
Benchmark Observations ... 21
Paginated Range Query Benchmark ... 22
Benchmark Results ... 22
Benchmark Observations ... 24
Paginated Rich Query Benchmark ... 25
Benchmark Results ... 25
Benchmark Observations ... 26
Submit Transaction Benchmark Results ... 28
Create Asset Benchmark ... 28
Benchmark Results ... 29
Benchmark Observations ... 31
Batch Create Asset Benchmark ... 31
Benchmark Results ... 32
Benchmark Observations ... 34
Appendix ... 35
Machine Configuration ... 35
Tools ... 35
Resources ... 35

Preface

This report intended to provide key processing and performance characteristics to architects, systems programmers, analysts and programmers. For best use of the performance reports, the user should be familiar with the concepts and operation of Hyperledger Fabric.

Performance observations have been obtained from testing a Hyperledger Fabric JavaScript smart contract, driven by Fabric-SDK-Node clients via Hyperledger Caliper, a performance benchmark harness for Hyperledger blockchain solutions. During the benchmarking Hyperledger Caliper was configured to drive all smart contract transactions through a Hyperledger Fabric client gateway.

Notes

The performance information is obtained by measuring the transaction throughput for different types of smart contract transactions. The term “transaction” is used in a generic sense, and refers to any interaction with a smart contract, regardless of the complexity of the subsequent interaction(s) with the blockchain platform.

Measuring transaction throughput demonstrates potential transaction rates, and the impact of the relative cost of different Hyperledger Fabric Stub API calls.

The data contained in the reports was measured in a controlled environment, results obtained in other environments might vary. For more details on the environments used, see the resources at the end of this report.

The performance data cannot be compared across versions of Hyperledger Fabric, as testing hardware and environments may have changed significantly. The testing contents and processing methodologies may have also changed between performance reports, and so cannot be compared.

Benchmark Information

System Under Test

The test topology is given in Figure 1 below. All tests were performed on a single Softlayer machine with the specification given in the Appendix section.

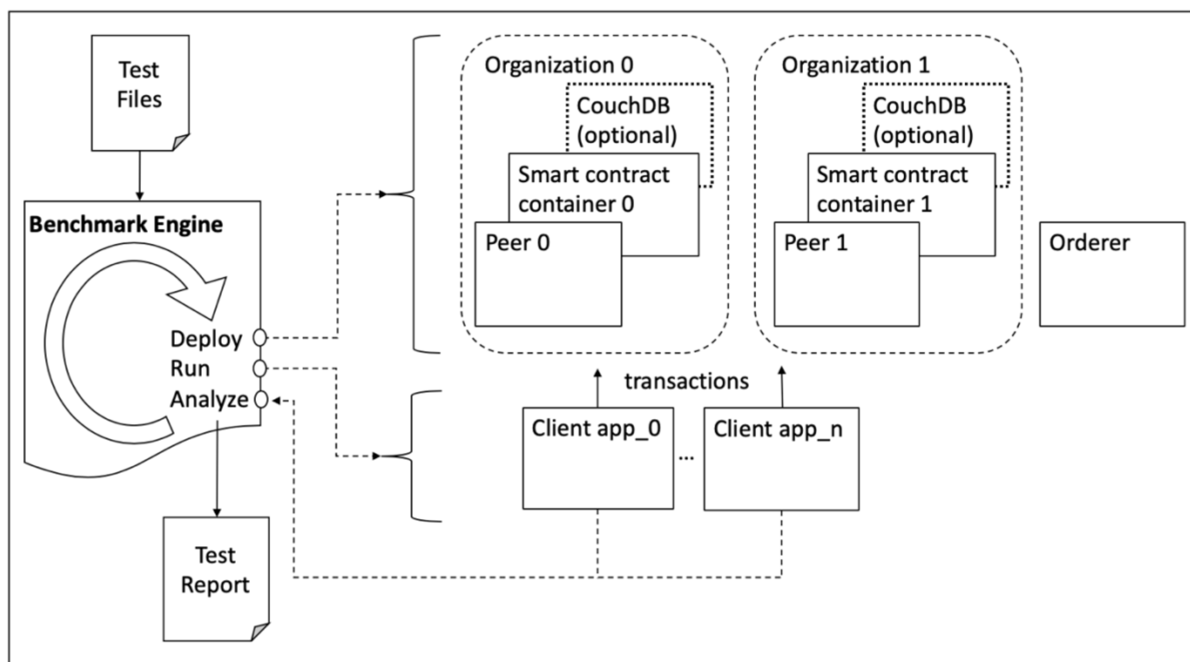


Figure 1: Test Topology

Two Hyperledger Fabric networks were investigated, each comprising of two organizations, with each organization having a single peer, and using a Solo ordering service. The difference between the two networks was the World State database used: one implemented LevelDB; the other CouchDB. The testing of distributed networks is the subject of future works.

The Smart Contract

All tests are facilitated by the `fixed-asset` smart contract that is deployed to the Hyperledger Fabric network. The smart contract facilitates the driving of core API methods that are commonly used by a smart contract developer.

Smart Contract Method	Description
emptyContract	Immediately returns an empty (null) response and represents the minimum possible overhead incurred through evaluation or submission of a smart contract method via a gateway.
createAsset	Performs a single `putState()` operation, inserting an asset of defined byte size into the World State database.
createAssetsFromBatch	Performs multiple `putState()` operations over an array of assets, inserting each into the World State database.
getAsset	Performs a single `getState()` operation, extracting and returning a single asset from the World State database using a passed UUID.

getAssetsFromBatch	Performs multiples `getState()` operations over an array of asset UUIDs, extracting and returning all asset from the World State database.
paginatedRangeQuery	Performs a `getStateByRangeWithPagination()` operation, based on passed start/end keys, a desired page size and passed bookmark. The records obtained from the query are processed and returned in a JSON response that also includes a new bookmark.
paginatedRichQuery	Performs a `getQueryResultWithPagination()` operation, based on a passed Mango query string, a desired page size and bookmark. The records obtained from the query are processed and returned in a JSON response that also includes a new bookmark. Only valid for deployments including a CouchDB World State database.

Smart contract methods may be evaluated or submitted via a Fabric Network gateway. An overview of possible transaction pathways from a client application interacting with Hyperedger Fabric is presented in Figure 2.

Evaluation of a smart contract method will not include interaction with the ordering service, and consequently will not result in appending to the leger; submission of a smart contract will result on the method being run on Hyperledger Fabric Peers as required by the endorsement policy and appended to the ledger by the ordering service.

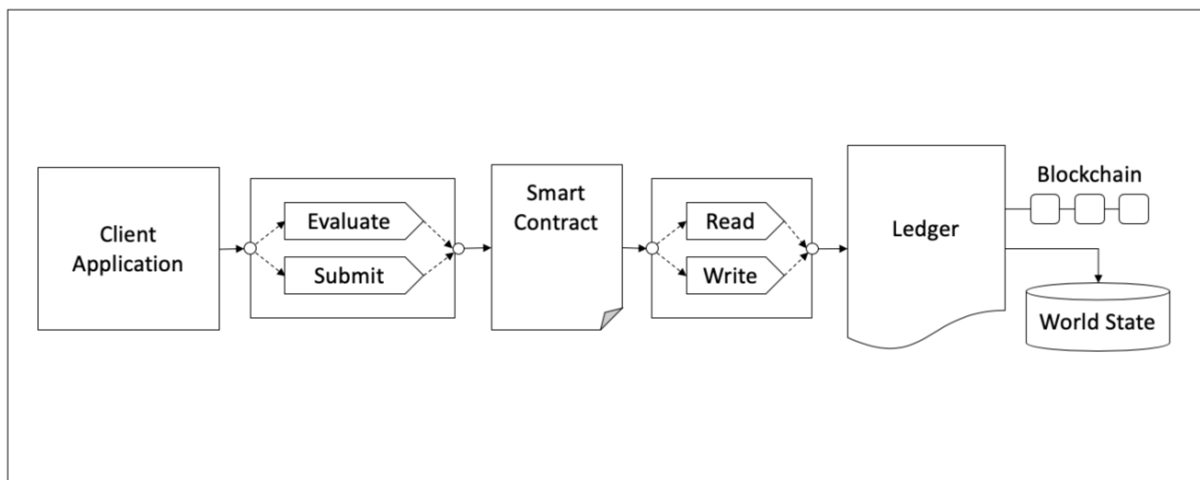


Figure 2: Possible Transaction Pathways

Smart Contract Benchmarks

The complete output of the benchmark runs, and the resources used to perform them, are in the resources section of the Appendix. All benchmarks are driven at maximum possible TPS for a duration of 5 minutes by 4 clients. This is followed by a driving the benchmarks at a set TPS for a duration of 5 minutes by 4 clients to enable resource utilization comparisons. The benchmarks comprise of:

Benchmark	Config File(s)	Description
Empty Contract	empty-contract-1of.yaml empty-contract-2of.yaml	Evaluates and submits \`emptyContract\` gateway transactions for the fixed-asset smart contract. This transaction performs no action. Repeated for different Endorsement Policies.
Create Asset	create-asset.yaml	Submits \`createAsset\` gateway transactions for the fixed-asset smart contract. Each transaction inserts a single asset into the world state database. Successive rounds increase the asset byte size inserted into the world state database.
Create Asset Batch	create-asset-batch.yaml	Submits \`createAssetsFromBatch\` gateway transactions for the fixed-asset smart contract. Each transaction inserts a sequence of fixed size assets into the world state database. Successive rounds increase the batch size of assets inserted into the world state database.
Get Asset	get-asset.yaml	Evaluates \`getAsset\` gateway transactions for the fixed-asset smart contract. Each transaction retrieves a single asset from the world state database. Successive rounds increase the asset byte size retrieved from the world state database.
Get Asset Batch	get-asset-batch.yaml	Evaluates \`getAssetsFromBatch\` gateway transactions for the fixed-asset smart contract. Each transaction retrieves a series of assets from the world state database. Successive rounds increase the batch size of assets retrieved from the world state database.
Paginated Range Query	mixed-range-query-pagination.yaml	Evaluates \`paginatedRangeQuery\` gateway transactions for the fixed-asset smart contract. Each transaction retrieves a set of assets from the world state database.

		Successive rounds increase the page size of assets retrieved from the world state database.
Paginated Rich Query	mixed-rich-query-pagination.yaml	Evaluates `paginatedRichQuery` gateway transactions for the fixed-asset smart contract. Each transaction retrieves a set of assets from the world state database. Successive rounds increase the page size of assets retrieved from the world state database.

Empty Contract Benchmark Results

The Empty Contract Benchmark consists of evaluating or submitting `emptyContract` gateway transactions for the fixed-asset smart contract deployed within LevelDB and CouchDB networks. This is repeated for networks that use the following endorsement policies:

- 1-of-any
- 2-of-any

Achievable throughput and associated latencies are investigated through maintaining a constant transaction backlog of 15 transactions for each of the 4 clients.

Resource utilization is investigated for fixed TPS rates of 750 and 350TPS for evaluate and submit transaction respectively.

Evaluate

When evaluating `emptyContract` gateway transactions, there is no interaction with the ledger. This results in the transaction pathway as depicted in Figure 3

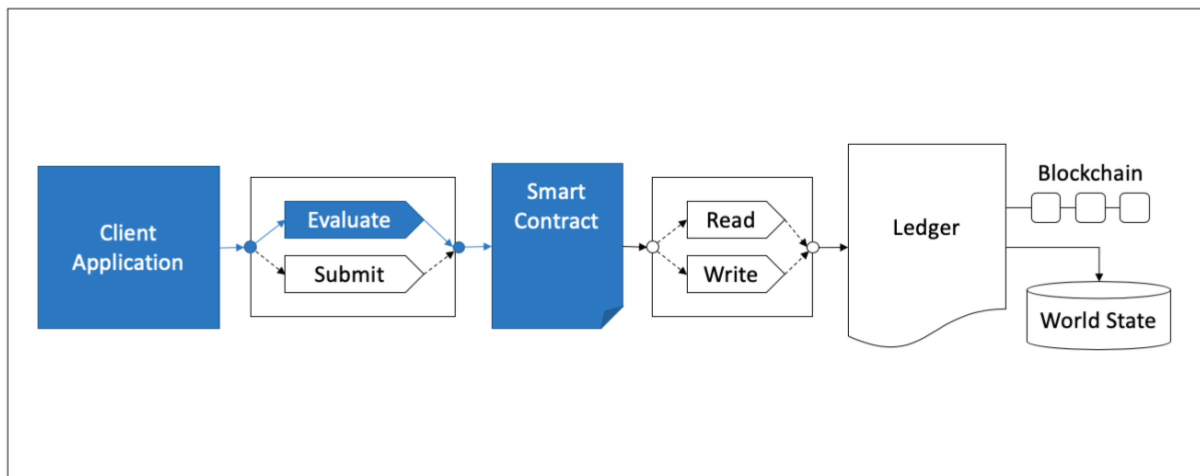


Figure 3: Evaluate Empty Contract Transaction Pathway

Evaluate Results

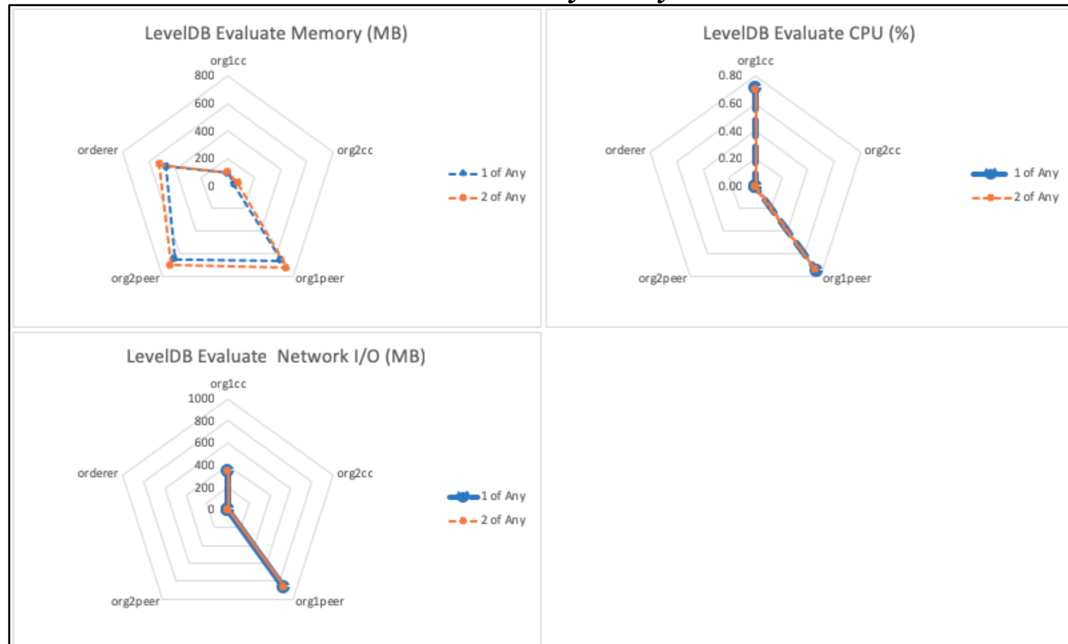
LevelDB- evaluate and submit transactions with varying endorsement policy

Type	Policy	Max Latency (s)	Avg Latency (s)	Throughput (TPS)
evaluate	1-of-any	0.18	0.04	792.3
evaluate	2-of-any	0.18	0.04	796.4

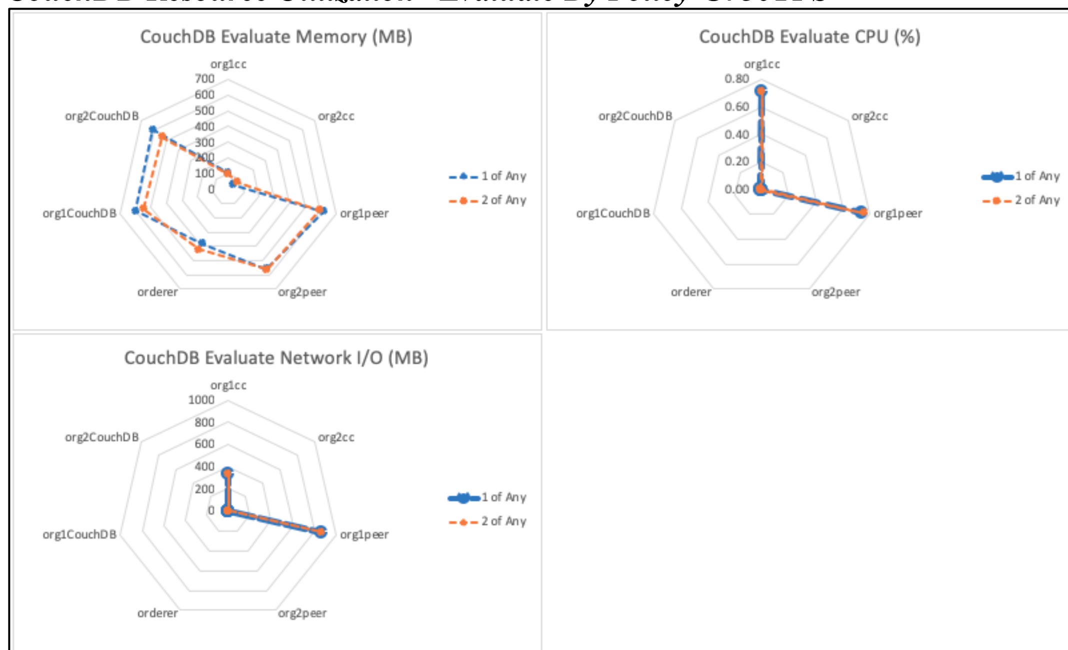
CouchDB- evaluate and submit transactions with varying endorsement policy

Type	Policy	Max Latency (s)	Avg Latency (s)	Throughput (TPS)
evaluate	1-of-any	0.16	0.04	789.9
evaluate	2-of-any	0.17	0.04	797.5

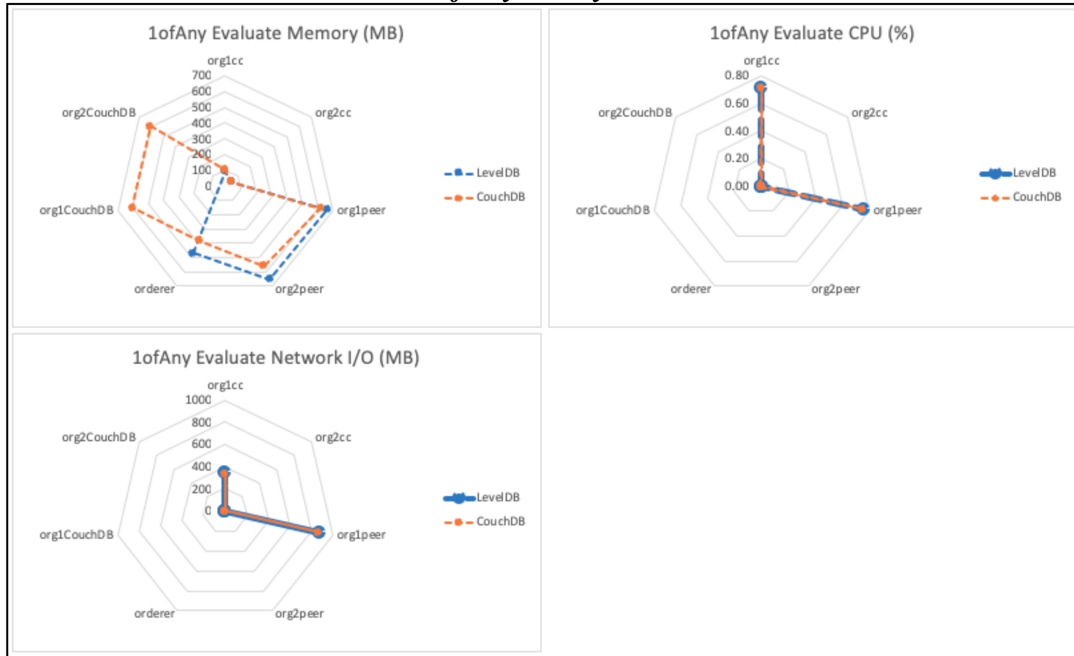
LevelDB Resource Utilization– Evaluate By Policy @750TPS



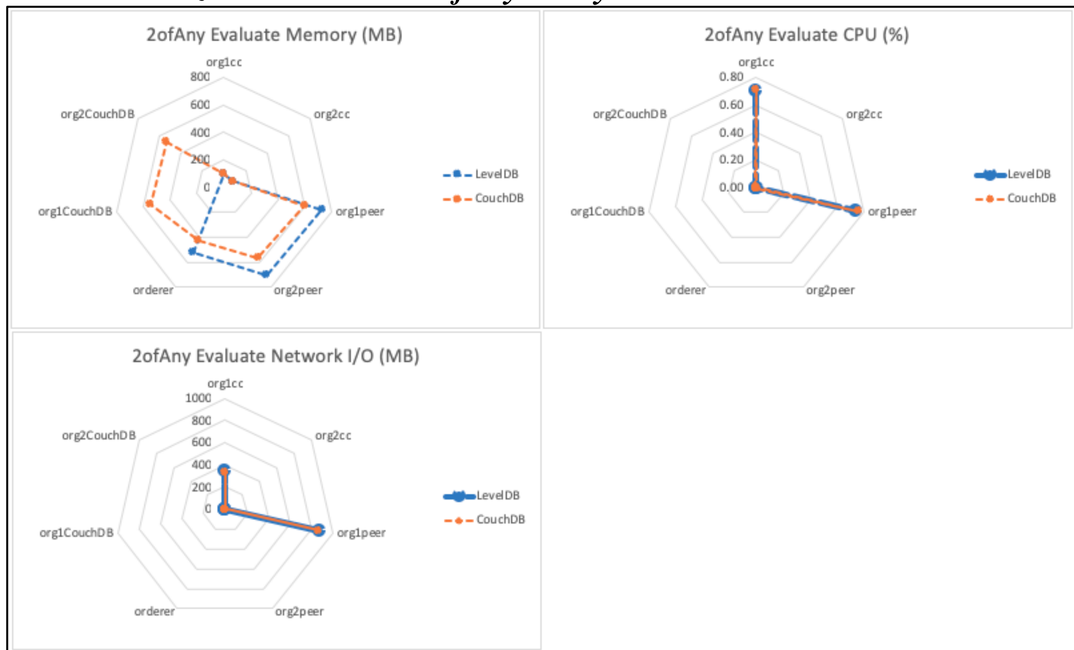
CouchDB Resource Utilization– Evaluate By Policy @750TPS



Resource Utilization– Evaluate 1ofAny Policy @750TPS



Resource Utilization– Evaluate 2ofAny Policy @750TPS



Evaluate Observations

With a fixed world state database, the endorsement policy has no impact on the consumed resources when evaluating gateway transactions.

In comparing a LevelDB world state database with a CouchDB equivalent, there is no appreciable difference in the achievable transaction throughput or transaction latency, nor the CPU or network I/O consumed by either implementation when varying the endorsement policy. There is a slight cost in additional memory requirements for the use of a CouchDB world state store.

Submit

When submitting `emptyContract` gateway transactions, the interaction is recorded on the ledger. This results in the transaction pathway as depicted in Figure 4.

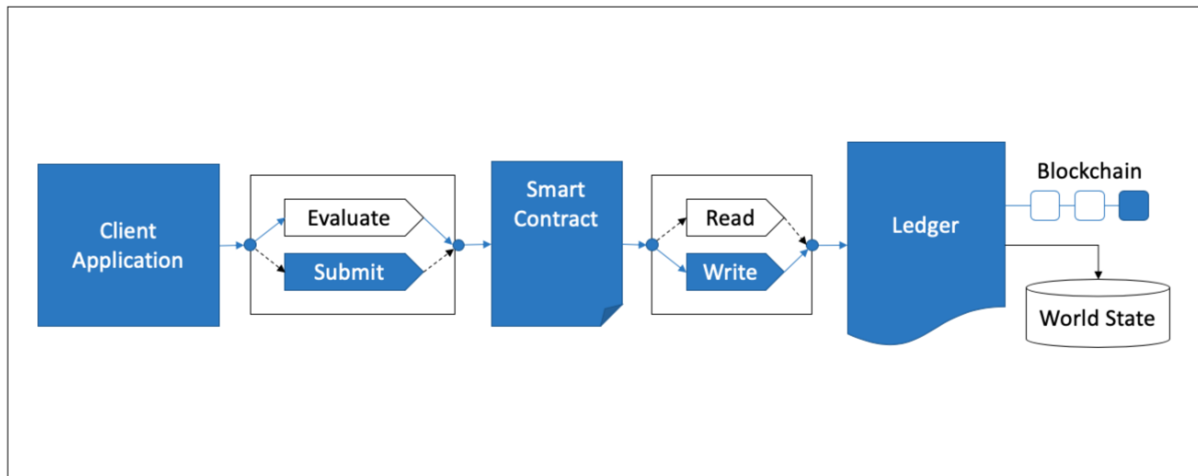


Figure 4: Submit Empty Contract Transaction Pathway

Submit Results

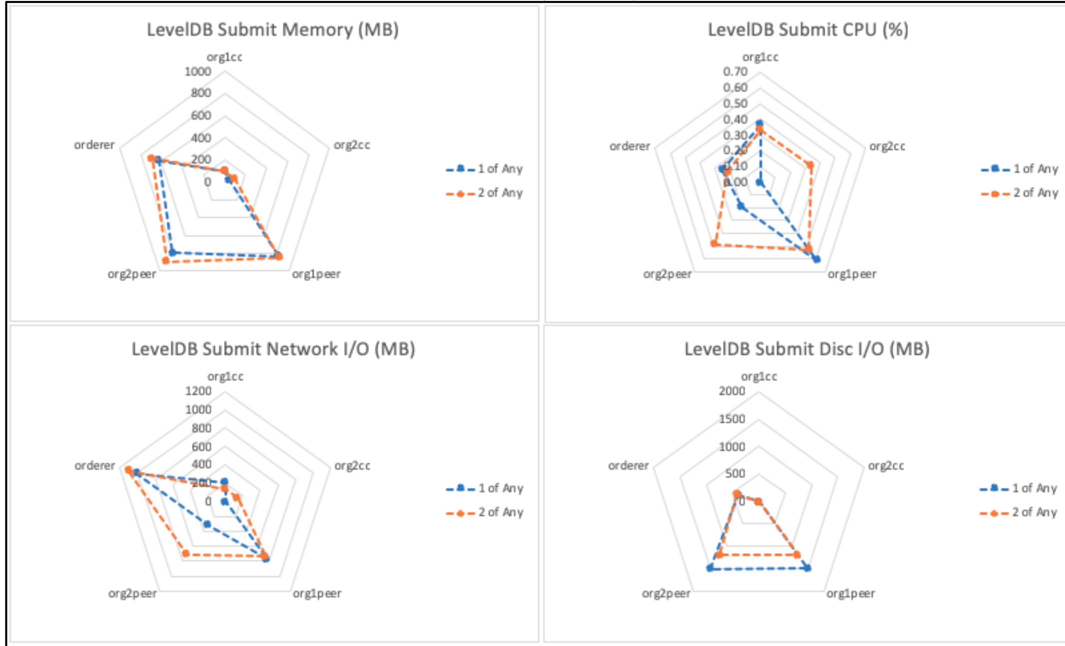
LevelDB- evaluate and submit transactions with varying endorsement policy

Type	Policy	Max Latency (s)	Avg Latency (s)	Throughput (TPS)
submit	1-of-any	0.41	0.09	485.4
submit	2-of-any	0.33	0.10	420.0

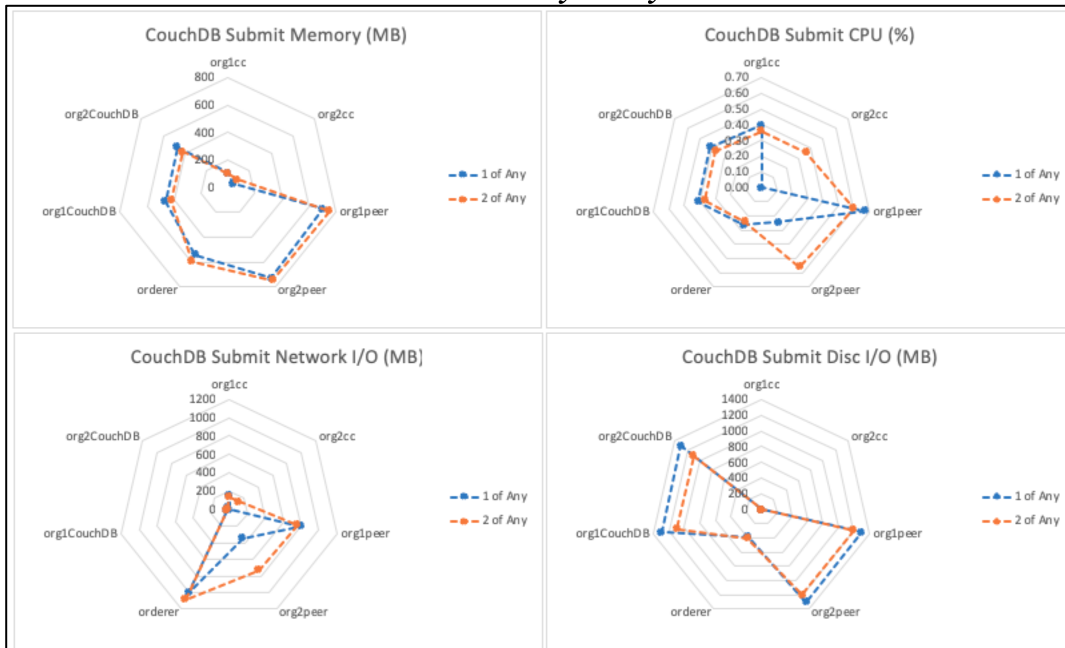
CouchDB- evaluate and submit transactions with varying endorsement policy

Type	Policy	Max Latency (s)	Avg Latency (s)	Throughput (TPS)
submit	1-of-any	0.52	0.11	380.5
submit	2-of-any	0.32	0.13	338.7

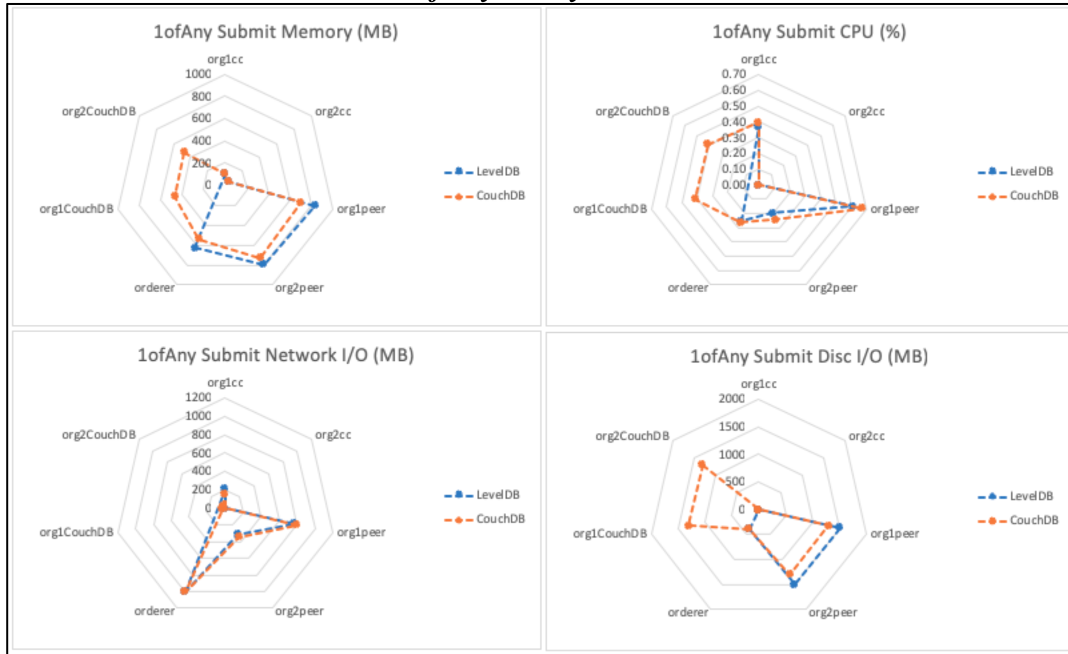
LevelDB Resource Utilization– Submit By Policy @350TPS



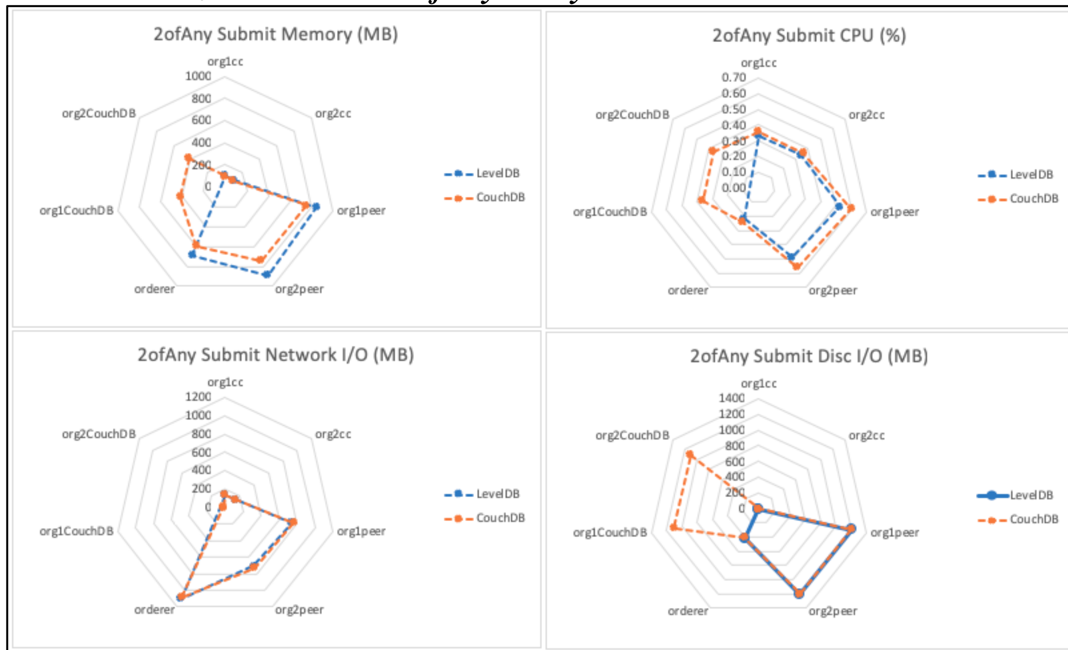
CouchDB Resource Utilization– Submit By Policy @350TPS



Resource Utilization– Submit 1ofAny Policy @350TPS



Resource Utilization– Submit 2ofAny Policy @350TPS



Submit Observations

LevelDB is observed to be beneficial for achievable throughput and reduced latencies in comparison to CouchDB during submission of an `emptyContract` gateway transaction for both investigated endorsement policies.

With a fixed world state database, the endorsement policy is observed to impact the consumed resources when submitting a transaction. Increasing the number of required endorsements is observed to increase the CPU and network I/O, through inclusion of additional peers and smart contract containers required to participate in each transaction.

In comparing a LevelDB world state database with a CouchDB equivalent, only the network I/O is observed to be equivalent when varying the endorsement policy. There is an observed penalty in additional memory, CPU and disc I/O requirements for the use of a CouchDB world state for the network as a whole, though the memory requirements of the peers are reduced.

Evaluate Transaction Benchmark Results

The following section focusses on the evaluation of a transaction though a network gateway; evaluation of a smart contract will result on the method being run on a single Hyperledger Fabric Peer and will not result in any interaction with the Orderer. The investigated scenarios are targeted at reading from the world state database, resulting in the transaction pathway depicted in Figure 5.

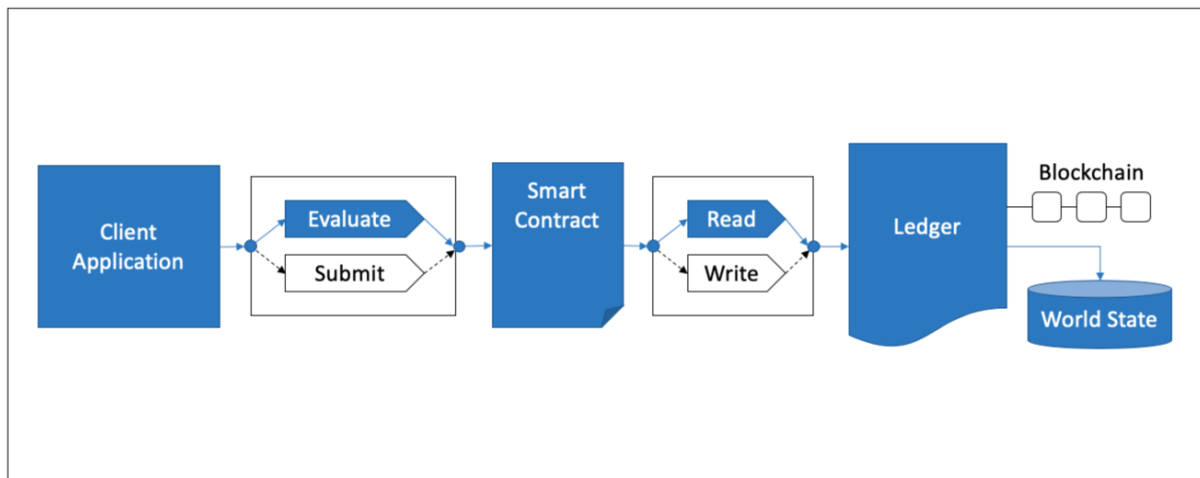


Figure 5: Evaluate Transaction Pathway

Get Asset Benchmark

Benchmark consists of evaluating `getAsset` gateway transactions for the fixed-asset smart contract deployed within LevelDB and CouchDB networks that uses a 2-of-any endorsement policy. Each transaction retrieves a single asset with a randomised UUID from the world state database.

Achievable throughput and associated latencies are investigated through maintaining a constant transaction backlog for each of the 4 clients. Successive rounds increase the size of the asset retrieved from the world state database.

Resource utilization is investigated for a fixed transaction rate of 350TPS, retrieving assets of size 8Kb.

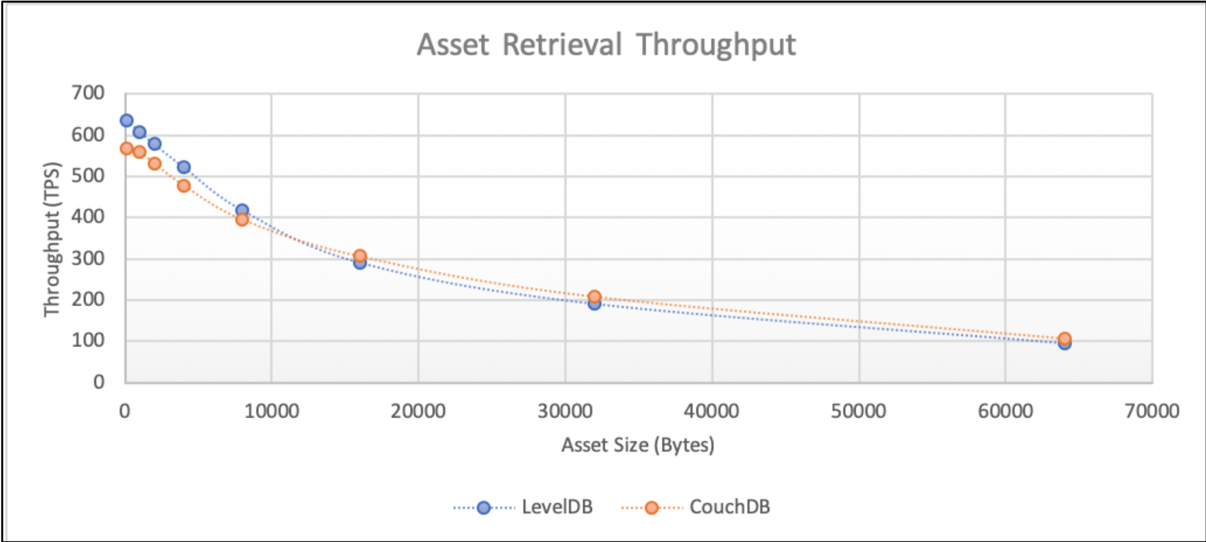
Benchmark Results

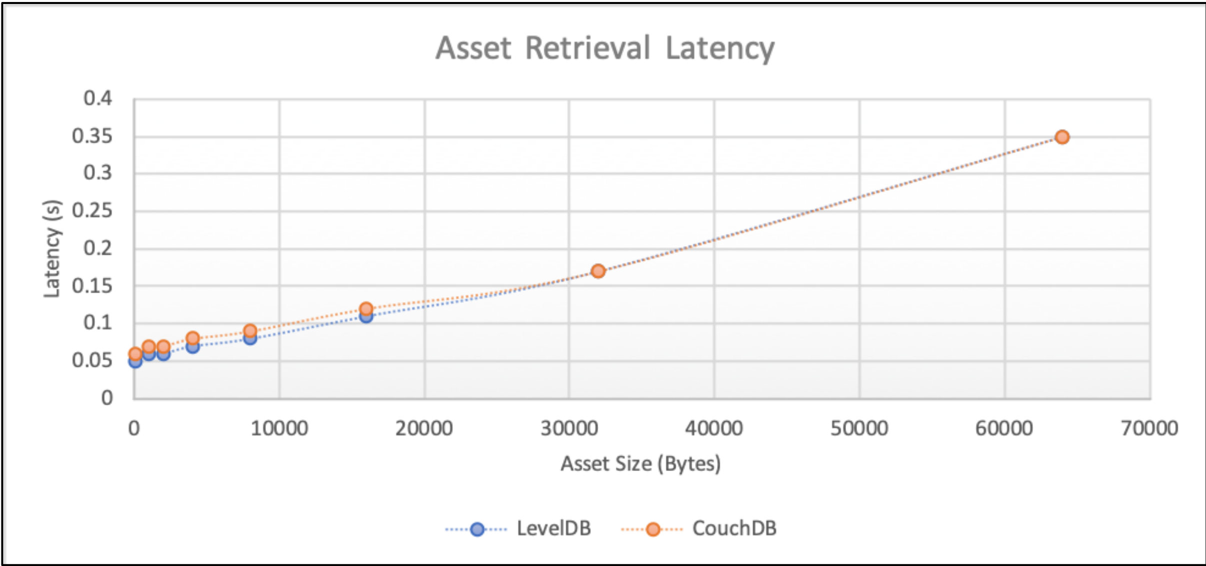
LevelDB

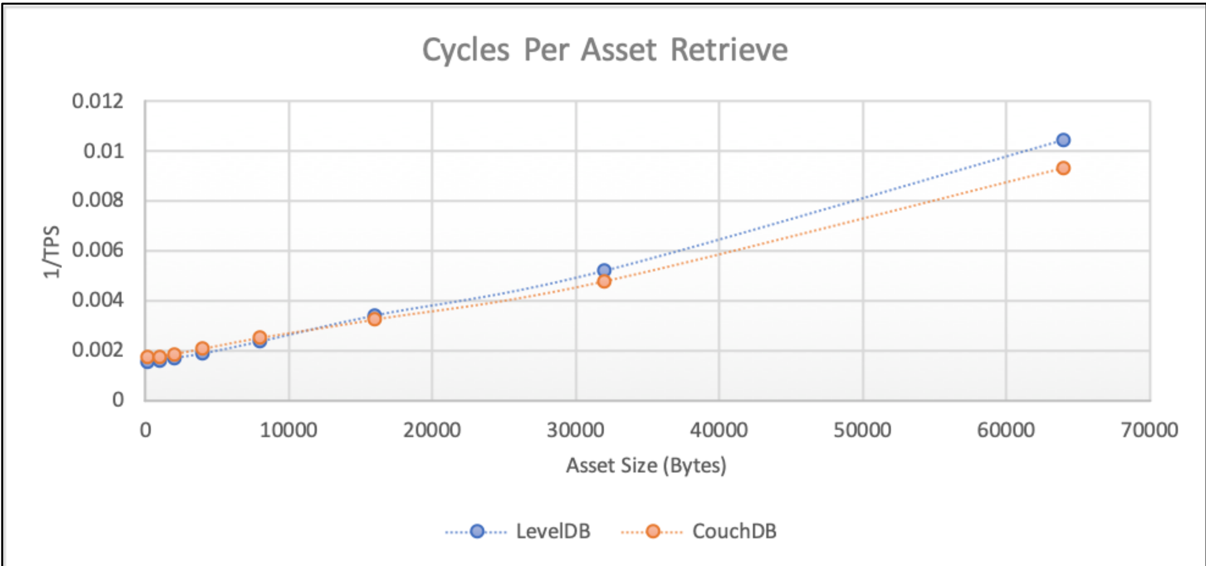
Asset Size (bytes)	Max Latency (s)	Avg Latency (s)	Throughput (TPS)
100	0.34	0.05	636.0
1k	0.21	0.06	611.1
2k	0.23	0.06	579.8
4k	0.20	0.07	516.8
8k	0.19	0.08	423.1
16k	0.24	0.11	293.6
32k	0.35	0.18	186.5
64k	0.73	0.35	96.0

CouchDB

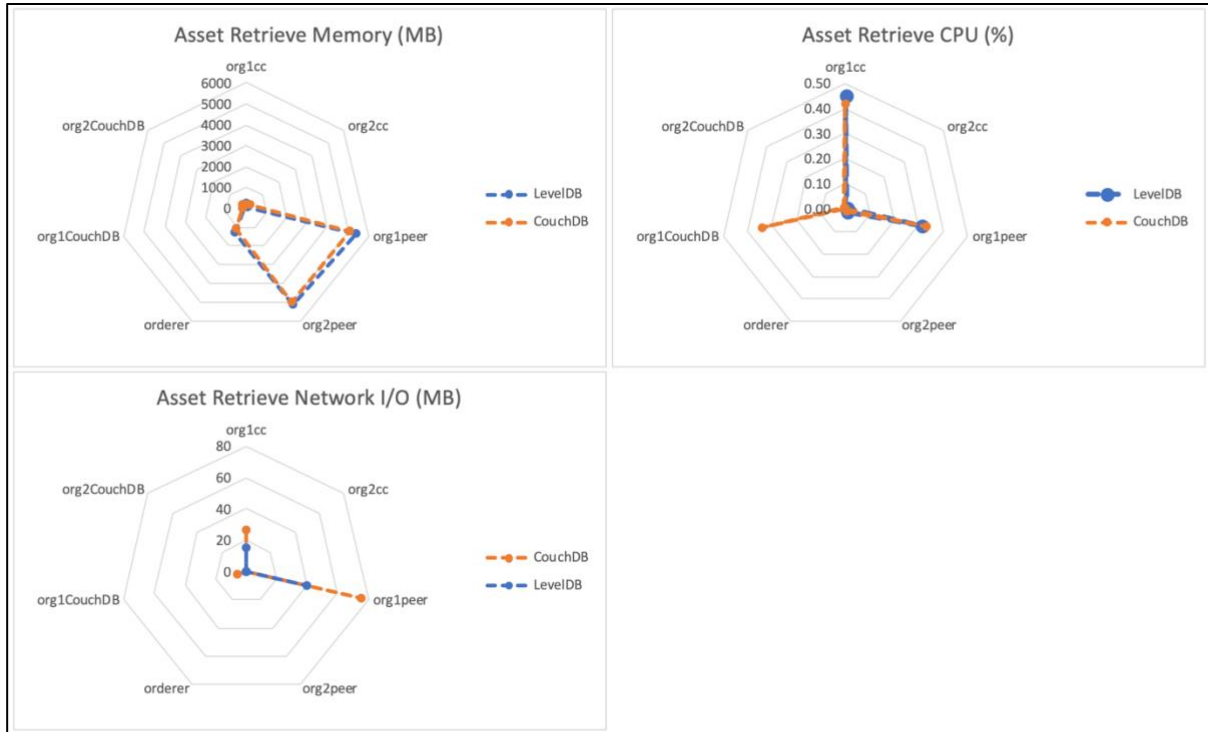
Asset Size (bytes)	Max Latency (s)	Avg Latency (s)	Throughput (TPS)
100	1.10	0.06	567.4
1K	1.06	0.07	558.9
2K	0.24	0.07	531.4
4K	0.25	0.08	478.0
8K	0.26	0.09	395.4
16K	0.29	0.12	306.1
32K	0.36	0.17	208.3
64K	0.75	0.35	107.0


Asset Retrieval Throughput
Throughput (TPS)
700
600
500
400
300
200
100
0
0
10000
20000
30000
40000
50000
60000
70000
Asset Size (Bytes)
LevelDB
CouchDB


Asset Retrieval Latency
Latency (s)
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
0
10000
20000
30000
40000
50000
60000
70000
Asset Size (Bytes)
LevelDB
CouchDB


Cycles Per Asset Retrieve
1/TPS
0.012
0.01
0.008
0.006
0.004
0.002
0
0
10000
20000
30000
40000
50000
60000
70000
Asset Size (Bytes)
LevelDB
CouchDB

Resource Utilization- 8k Assets @350TPS



Benchmark Observations

The CouchDB world state database is observed to achieve comparable throughput and lower latencies than a LevelDB equivalent, with higher achievable TPS for assets that are larger than 10Kb.

In comparing a LevelDB world state database with a CouchDB equivalent during asset retrieval, both consume similar memory resources, though the CouchDB world state database results in greater network I/O and a CPU overhead for the CouchDB instance that is not offset at the peer.

Batch Get Asset Benchmark

Benchmark consists of evaluating `getAssetsFromBatch` gateway transactions for the fixed-asset smart contract deployed within LevelDB and CouchDB networks that uses a 2-of-any endorsement policy. Each transaction retrieves a set of assets, formed by a randomised selection of available UUIDs, from the world state database.

Achievable throughput and associated latencies are investigated through maintaining a constant transaction backlog for each of the 4 clients. Successive rounds increase the batch size of the assets retrieved from the world state database with a fixed asset size of 8Kb.

Resource utilization is investigated for a fixed transaction rate of 30TPS and a batch size of 20 assets, each of size 8Kb.

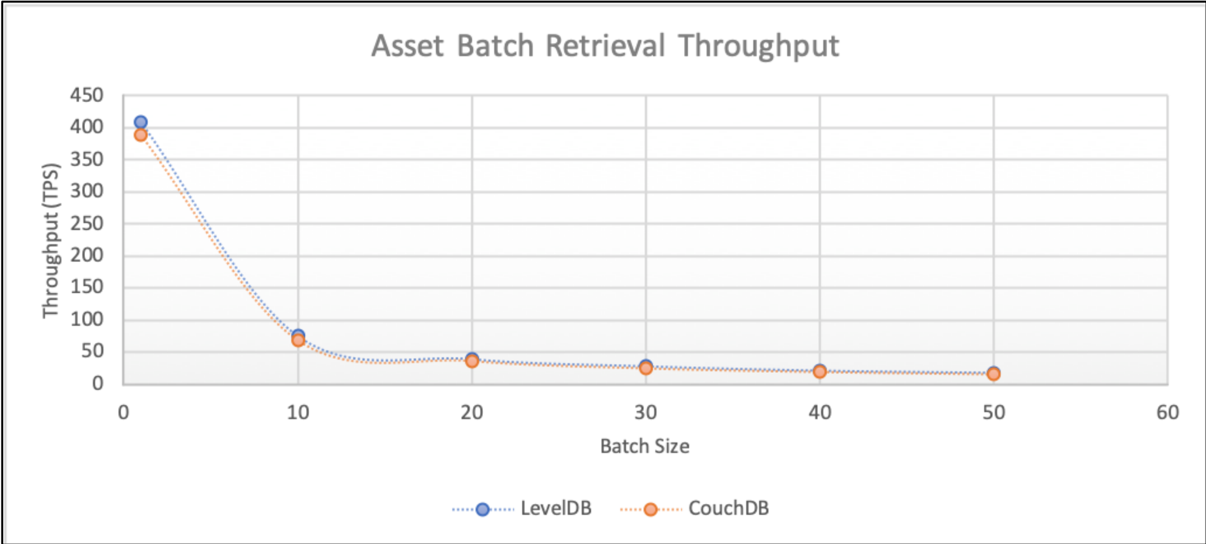
Benchmark Results

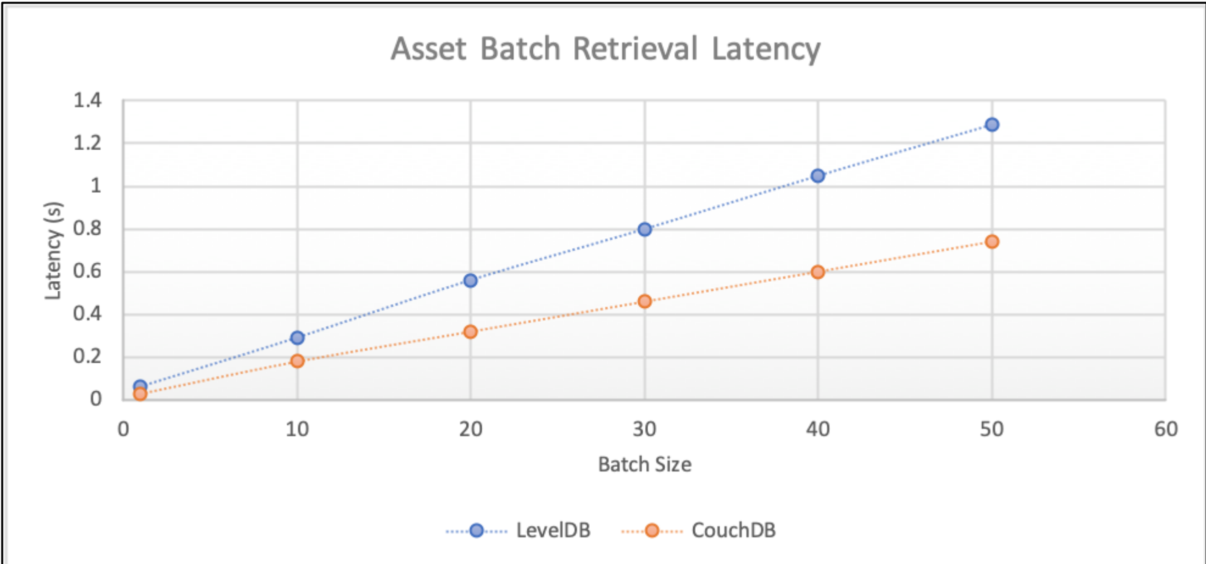
LevelDB

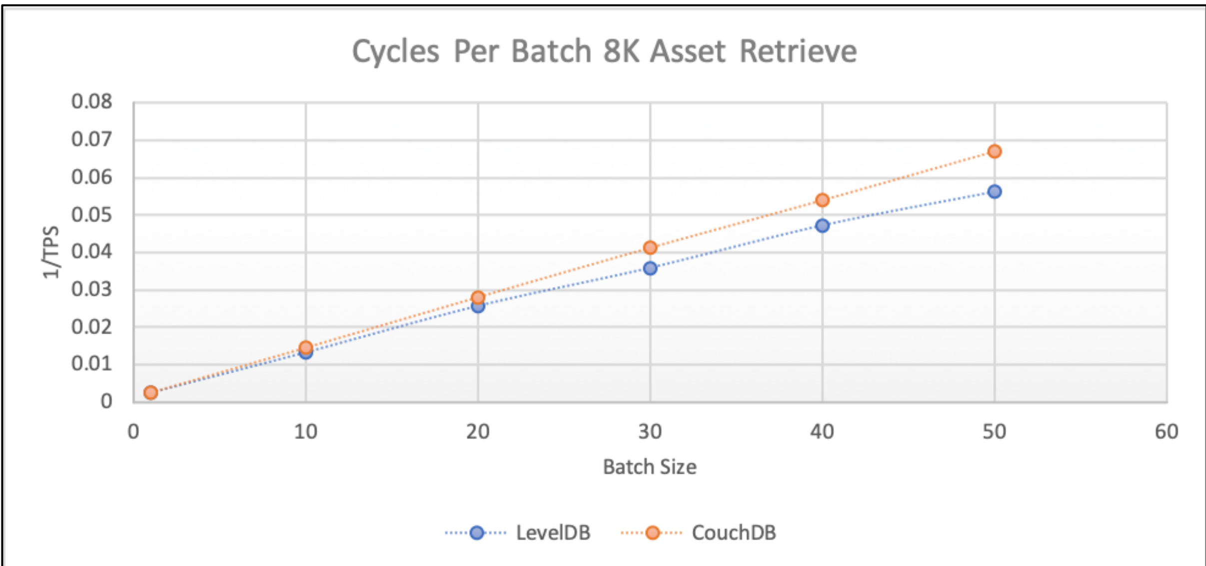
Batch Size	Max Latency (s)	Avg Latency (s)	Throughput (TPS)
1	0.20	0.06	408.7
10	0.48	0.29	75.5
20	1.03	0.56	39.0
30	1.34	0.80	27.9
40	1.68	1.05	21.2
50	2.14	1.29	17.8

CouchDB

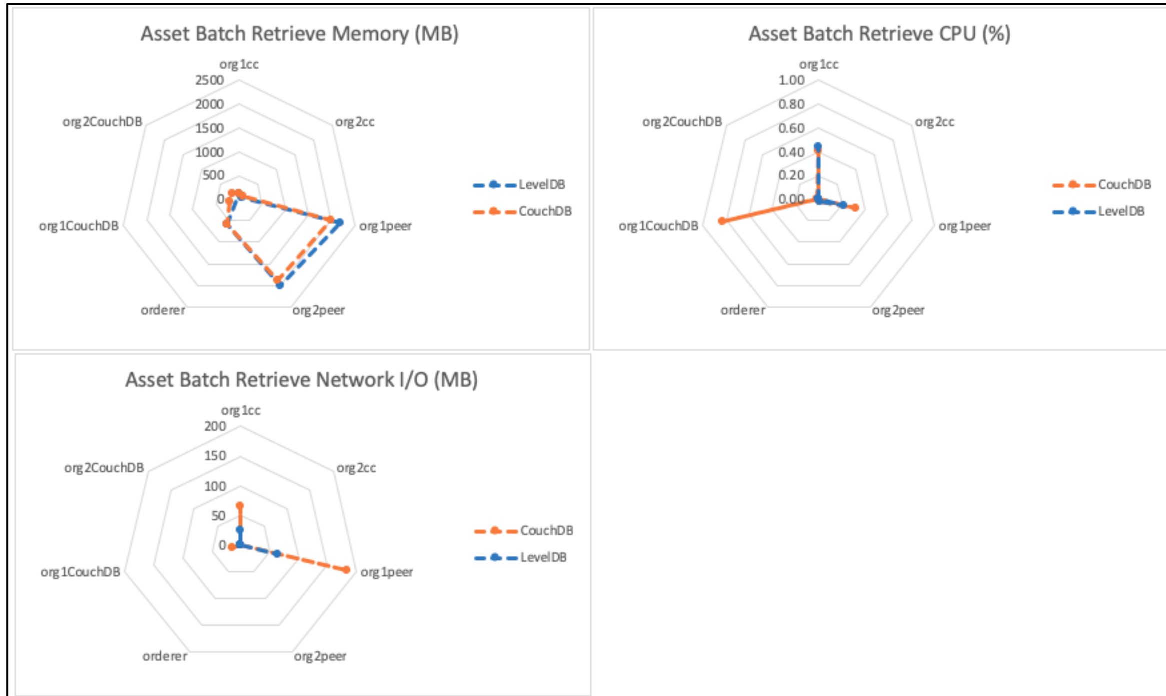
Batch Size	Max Latency (s)	Avg Latency (s)	Throughput (TPS)
1	0.15	0.03	388.9
10	0.46	0.18	68.5
20	0.64	0.32	35.6
30	0.84	0.46	24.2
40	1.10	0.60	18.5
50	1.32	0.74	14.9


Asset Batch Retrieval Throughput
Throughput (TPS)
450
400
350
300
250
200
150
100
50
0
0
10
20
30
40
50
60
Batch Size
LevelDB
CouchDB


Asset Batch Retrieval Latency
Latency (s)
1.4
1.2
1
0.8
0.6
0.4
0.2
0
0
10
20
30
40
50
60
Batch Size
LevelDB
CouchDB


Cycles Per Batch 8K Asset Retrieve
1/TPS
0.08
0.07
0.06
0.05
0.04
0.03
0.02
0.01
0
0
10
20
30
40
50
60
Batch Size
LevelDB
CouchDB

Resource Utilization- Batch Size 20 @30TPS



Benchmark Observations

Use of a LevelDB world state enables higher throughput compared to CouchDB, though this occurs with higher latencies for each transaction.

In comparing a LevelDB world state database with a CouchDB equivalent during batch retrieve, there are similarities with the `Get Asset Benchmark`: implementing a CouchDB incurs a greater CPU and network I/O cost without alleviating CPU utilization of the peer.

Paginated Range Query Benchmark

Benchmark consists of evaluating `paginatedRangeQuery` gateway transactions for the fixed-asset smart contract deployed within LevelDB and CouchDB networks that use a 2-of-any endorsement policy.

Each transaction retrieves a fixed number of mixed byte size assets in the range [100, 1000, 2000, 4000, 8000, 16000, 32000, 64000] from the world state database.

Achievable throughput and associated latencies are investigated through maintaining a constant transaction backlog for each of the 4 clients. Successive rounds increase the page size of assets retrieved from the world state database.

Resource utilization is investigated for a fixed transaction rate of 30TPS and a batch size of 20 assets.

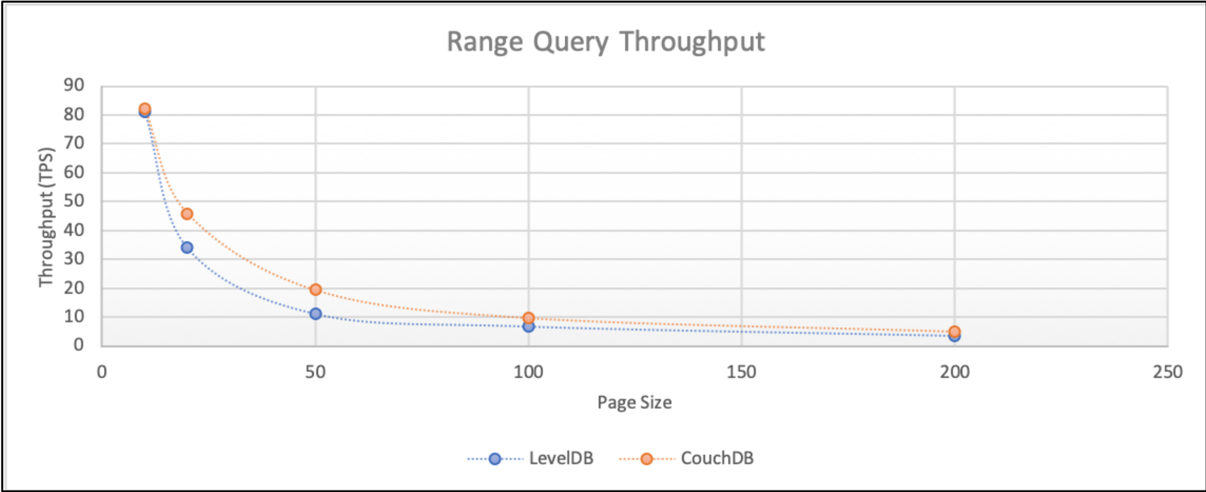
Benchmark Results

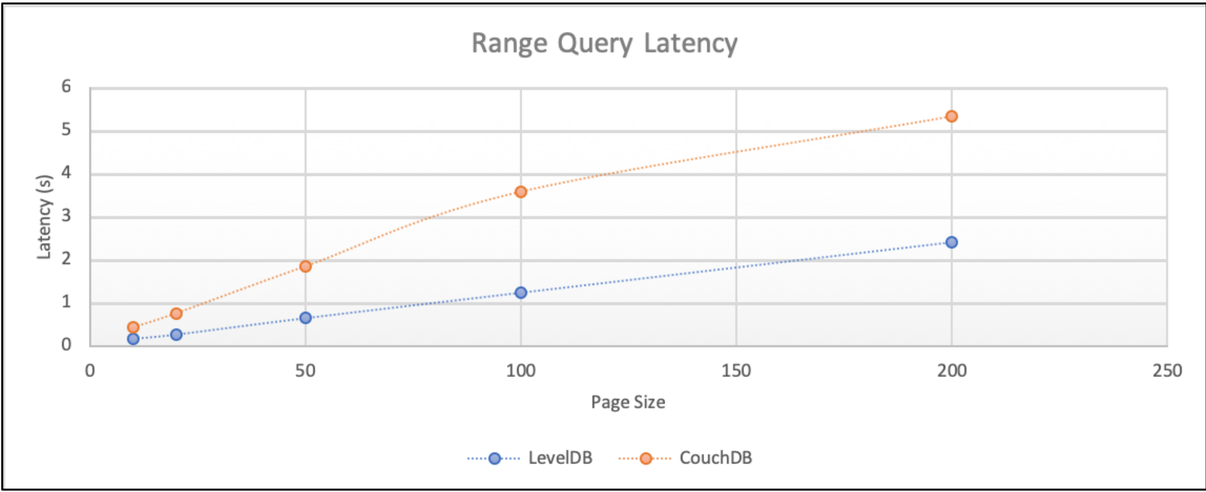
LevelDB

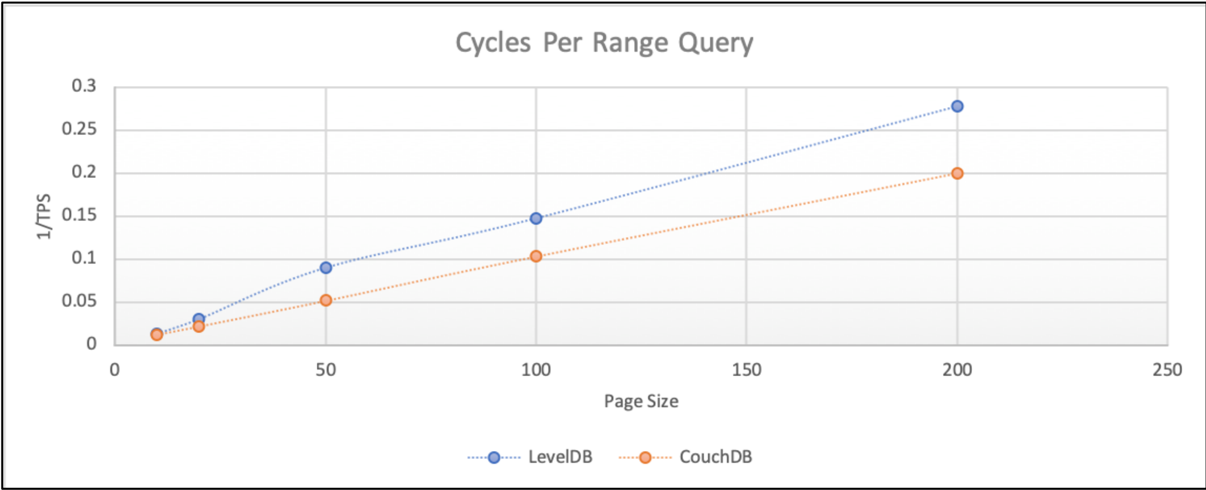
Page Size	Max Latency (s)	Avg Latency (s)	Throughput (TPS)
10	0.23	0.16	81.1
20	0.37	0.26	34.0
50	0.86	0.64	11.2
100	1.59	1.23	6.8
200	2.86	2.40	3.6
500	9.02	7.07	0.9

CouchDB

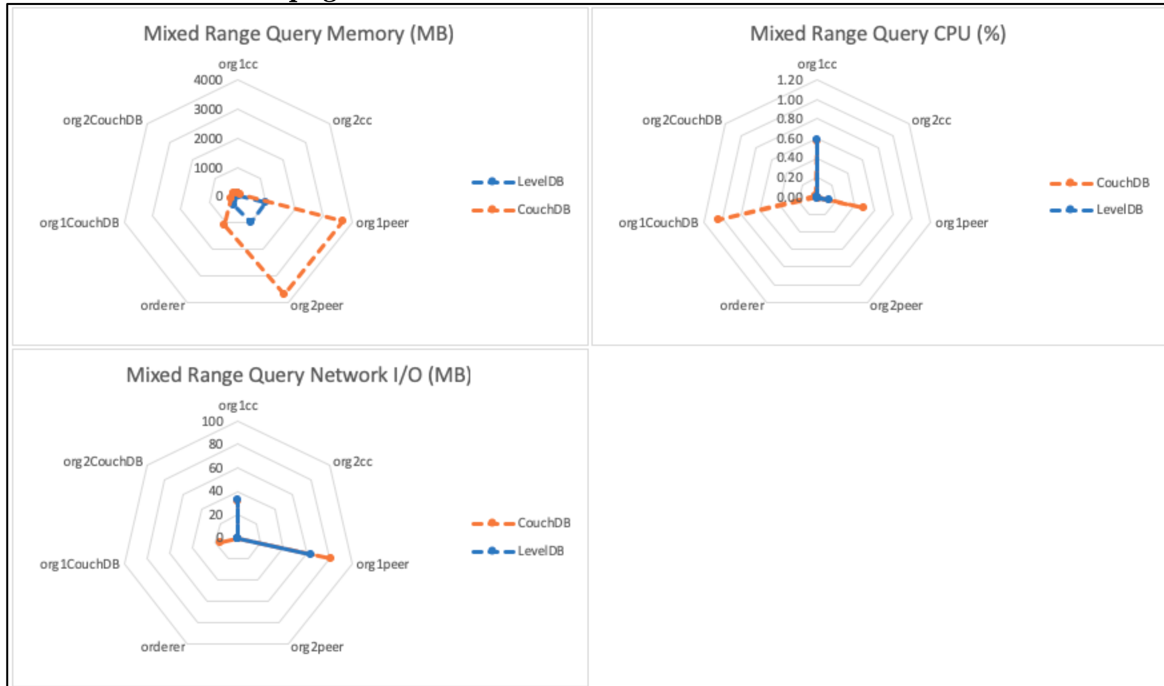
Page Size	Max Latency (s)	Avg Latency (s)	Throughput (TPS)
10	0.94	0.42	82.1
20	1.60	0.75	45.9
50	4.09	1.84	19.4
100	8.03	3.57	9.7
200	16.55	5.32	5.0
500	15.96	4.80	1.6


Range Query Throughput
Throughput (TPS)
90
80
70
60
50
40
30
20
10
0
0
50
100
150
200
250
Page Size
LevelDB
CouchDB


Range Query Latency
Latency (s)
6
5
4
3
2
1
0
0
50
100
150
200
250
Page Size
LevelDB
CouchDB


Cycles Per Range Query
1/TPS
0.3
0.25
0.2
0.15
0.1
0.05
0
0
50
100
150
200
250
Page Size
LevelDB
CouchDB

Resource Utilization- page size 20 @30TPS



Benchmark Observations

Use of a CouchDB world state database enables greater throughput but higher latencies than the LevelDB equivalent.

In comparing the resource utilization of a LevelDB world state database with a CouchDB equivalent during a range query, the CouchDB world state incurs a cost in memory, network I/O and CPU utilization. In particular, use of a CouchDB world state for a range query is observed to result in significant increases in CPU and memory utilization in the peer, with an associated increase in network I/O as a result of communication with the CouchDB instance.

When comparing the range query page sizes against a matching batch size in the `Get Asset Batch Benchmark`, it is observed to be more efficient to use a batch retrieval mechanism with known UUIDs.

Paginated Rich Query Benchmark

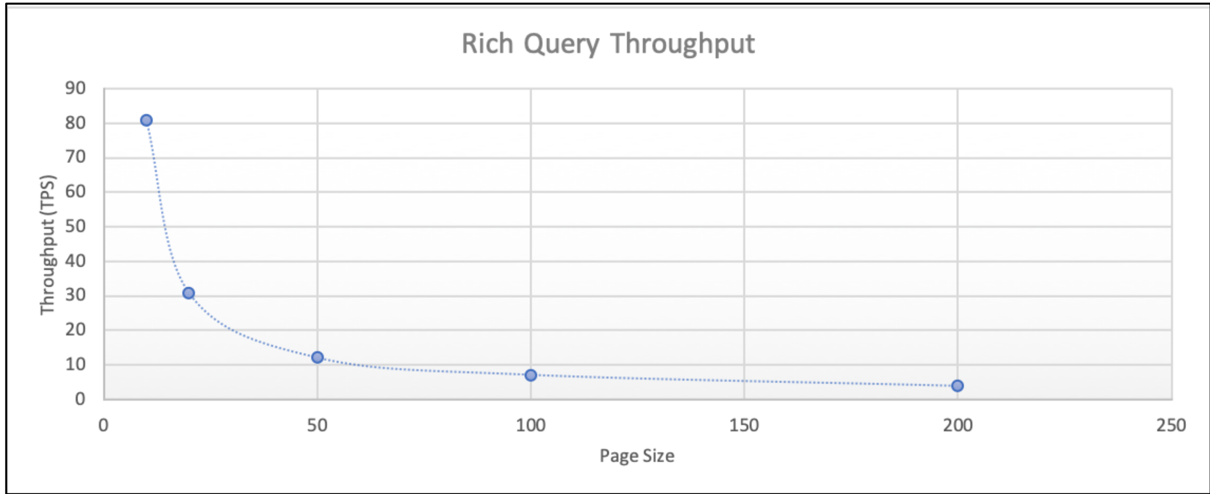
Benchmark consists of evaluating `paginatedRichQuery` gateway transactions for the fixed-asset smart contract deployed within a CouchDB network that uses a 2-of-any endorsement policy. Each transaction retrieves a fixed number of mixed byte size assets in the range [100, 1000, 2000, 4000, 8000, 16000, 32000, 64000] from the world state database based on the following Mango query that matches an index created in CouchDB:

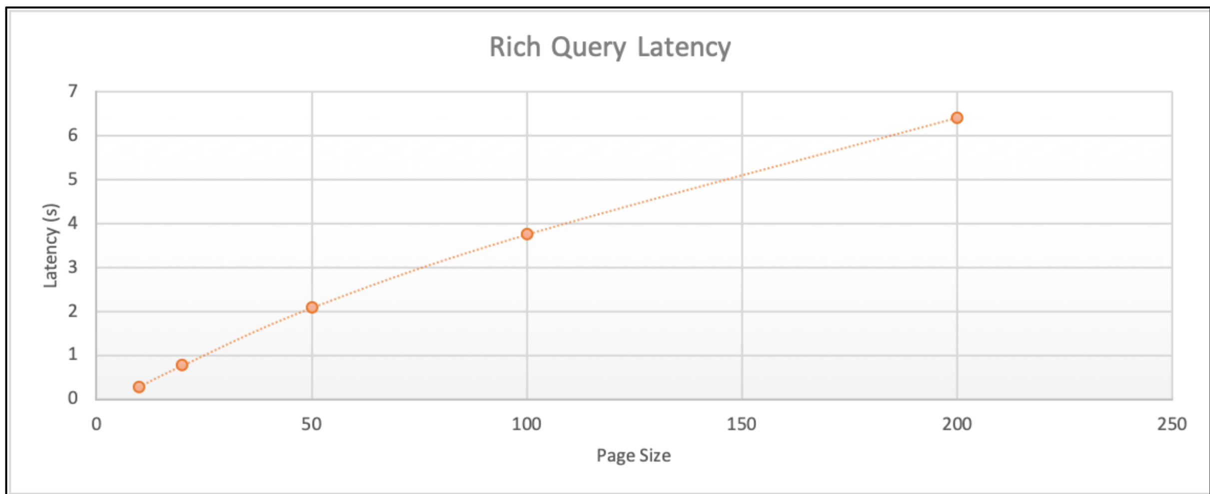
```
{
    'selector': {
        'docType': 'fixed-asset',
        'creator': 'clientId’,
        'bytesize': 'bytesize'
      }
}
```

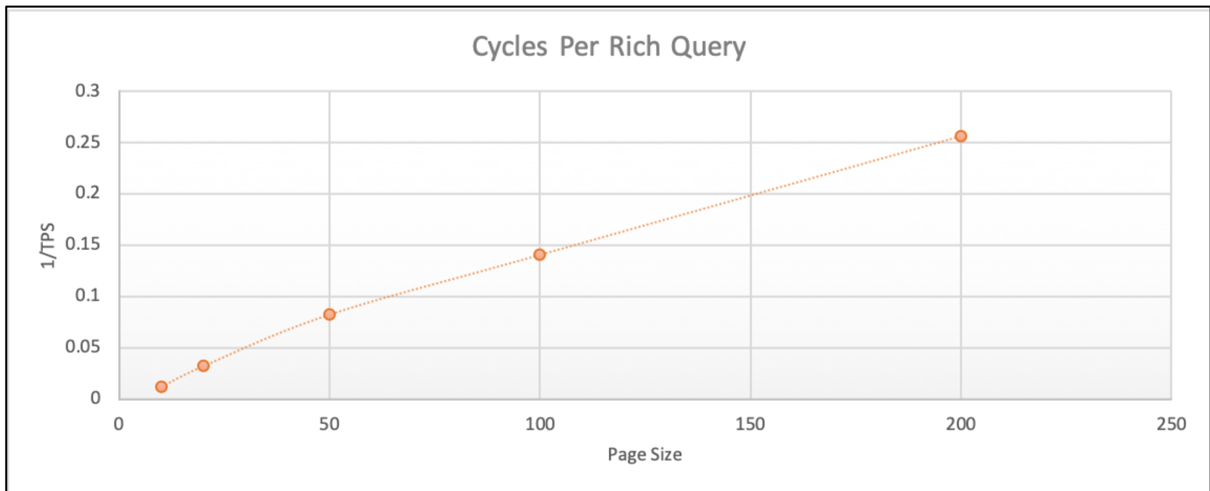
Achievable throughput and associated latencies are investigated through maintaining a constant transaction backlog for each of the 4 clients. Successive rounds increase the page size of assets retrieved from the world state database.

Benchmark Results

Page Size	Max Latency (s)	Avg Latency (s)	Throughput (TPS)
10	21.68	0.29	80.9
20	14.58	0.77	30.8
50	15.13	2.08	12.1
100	16.31	3.75	7.1
200	23.35	6.41	3.9
500	23.48	4.49	1.1







Benchmark Observations

Increasing the page size of a rich query has significant impact on the achievable throughput and latency. This corresponds with significantly increased network I/O across the target peer, smart contract and the CouchDB world state database.

Inspection of the resource utilization statistics for the individual benchmark runs show that the peer must deal with a significant network I/O load. This is a result of the peer obtaining and relaying the information from CouchDB to the smart contract transaction, and then passing back the resulting data from the smart contract transaction to the calling client application.

Submit Transaction Benchmark Results

The following section focusses on the submission of a transaction though a network gateway; submission of a smart contract will result on the method being run on Hyperledger Fabric Peers as required by the endorsement policy and appended to the ledger by the Orderer. The investigated scenarios are targeted at writing to the world state database, resulting in the transaction pathway as depicted in Figure 6.

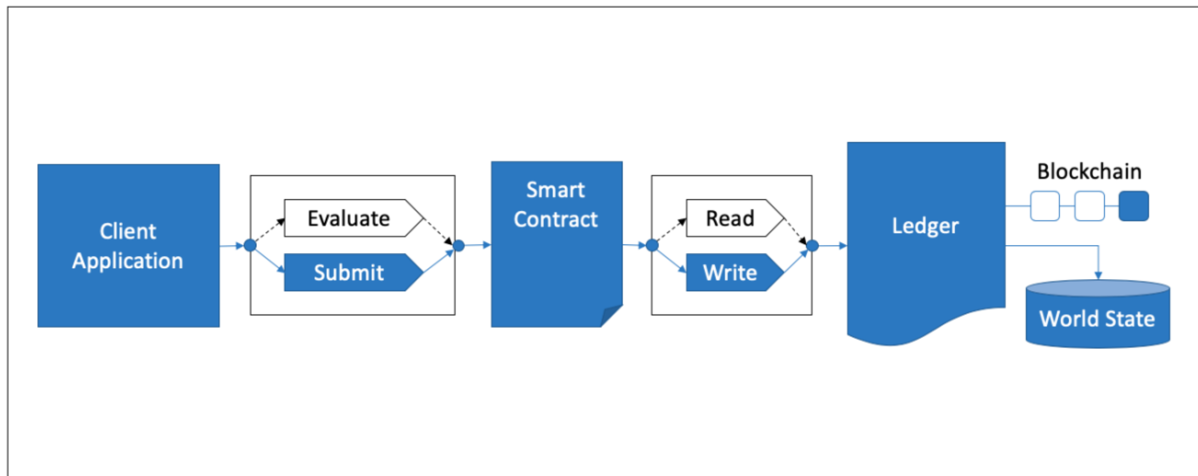


Figure 6: Submit Transaction Pathway

Create Asset Benchmark

Benchmark consists of submitting ` createAsset` gateway transactions for the fixed-asset smart contract deployed within LevelDB and CouchDB networks that uses a 2-of-any endorsement policy. Each transaction inserts a single asset into the world state database.

Achievable throughput and associated latencies are investigated through maintaining a constant transaction backlog for each of the 4 clients. Successive rounds increase the size of the asset inserted into the world state database.

Resource utilization is investigated for a fixed transaction rate of 125TPS and an asset size of 8Kb.

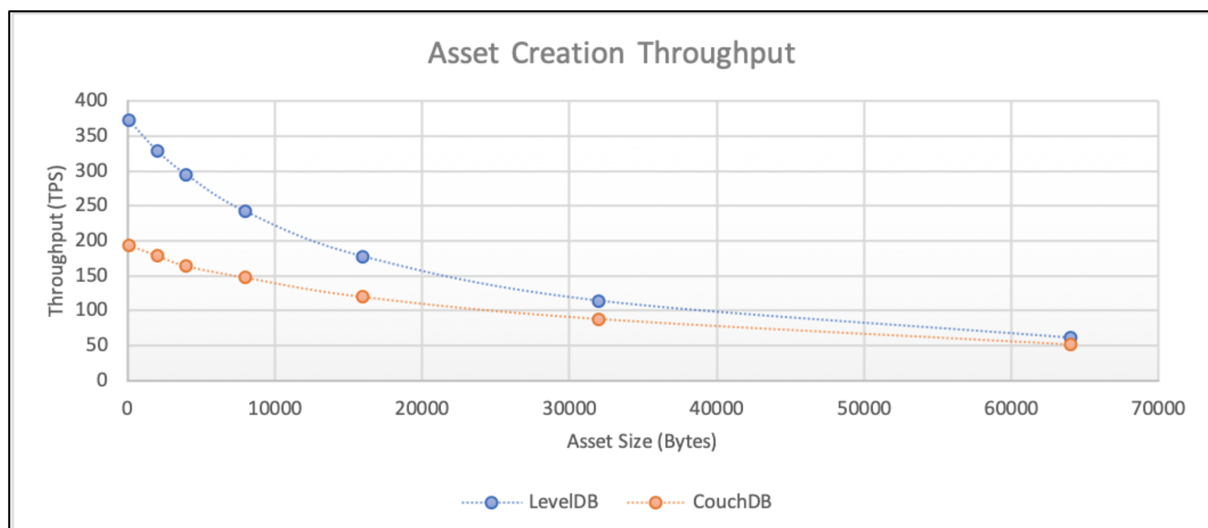
Benchmark Results

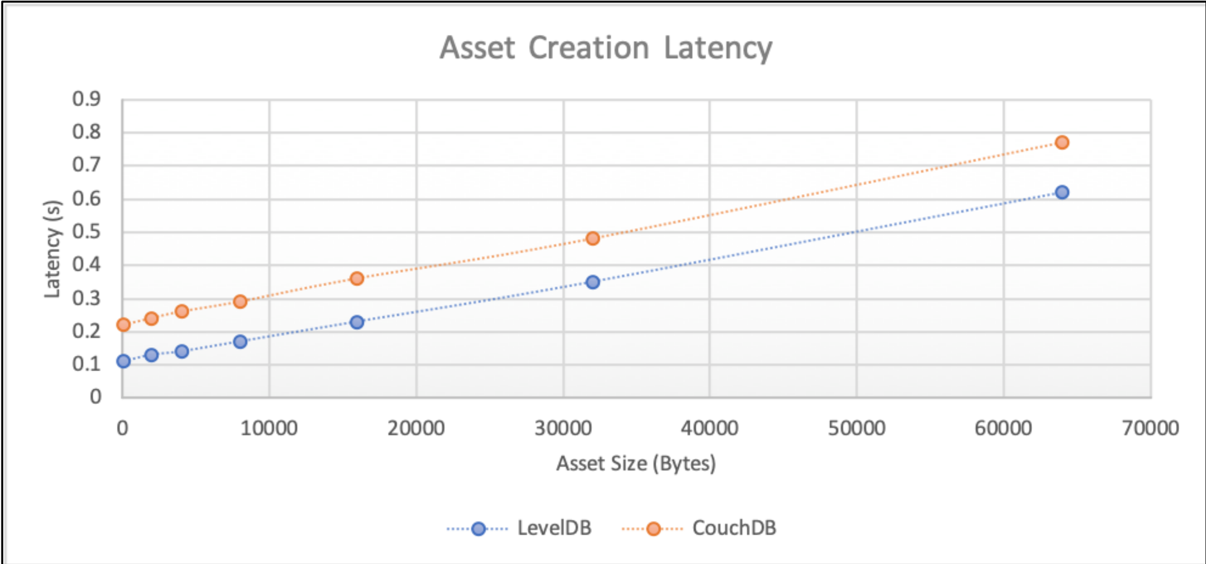
Level DB

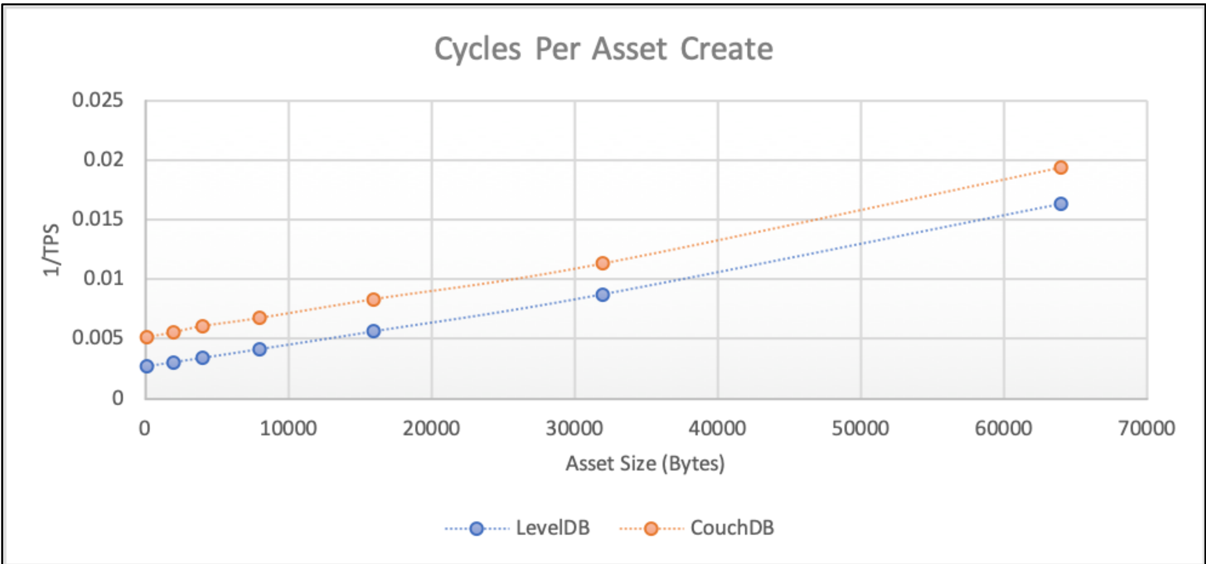
Asset Size (bytes)	Max Latency (s)	Avg Latency (s)	Throughput (TPS)
100	0.48	0.11	372.5
2k	0.55	0.13	329.2
4k	0.60	0.14	294.7
8k	0.71	0.17	242.0
16k	0.76	0.23	177.6
32k	0.95	0.35	114.3
64k	1.45	0.62	61.2

CouchDB

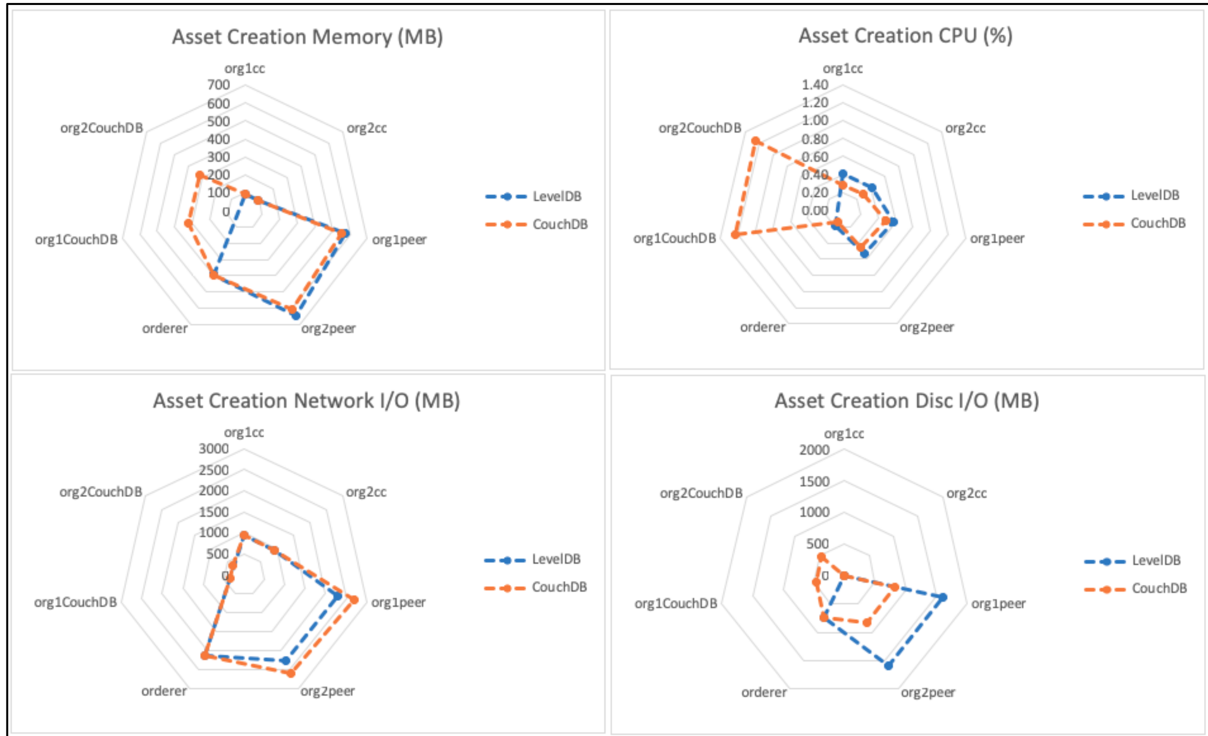
Asset Size (bytes)	Max Latency (s)	Avg Latency (s)	Throughput (TPS)
100	0.71	0.22	194.0
2K	0.57	0.24	179.2
4K	0.54	0.26	164.1
8K	0.74	0.29	147.7
16K	0.88	0.36	119.9
32K	0.99	0.48	88.3
64K	1.58	0.77	51.7




Asset Creation Latency
Latency (s)
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
10000
20000
30000
40000
50000
60000
70000
Asset Size (Bytes)
LevelDB
CouchDB


Cycles Per Asset Create
1/TPS
0.025
0.02
0.015
0.01
0.005
0
0
10000
20000
30000
40000
50000
60000
70000
Asset Size (Bytes)
LevelDB
CouchDB

Resource Utilization- 8k Assets @125TPS



Benchmark Observations

LevelDB facilitates asset addition at higher TPS and lower latencies than CouchDB. The throughput advantage of LevelDB is lessened with large asset sizes, but the latency advantage is retained.

In comparing the resource utilization of a LevelDB world state database with a CouchDB equivalent during asset creation, a CouchDB world state is CPU intensive, but is beneficial in terms of disc I/O.

Batch Create Asset Benchmark

Benchmark consists of submitting `createAssetsFromBatch` gateway transactions for the fixed-asset smart contract deployed within LevelDB and CouchDB networks that uses a 2-of-any endorsement policy. Each transaction inserts a set of assets into the world state database.

Achievable throughput and associated latencies are investigated through maintaining a constant transaction backlog for each of the 4 clients. Successive rounds increase the batch size of the assets inserted into the world state database with a fixed asset size of 8Kb.

Resource utilization is investigated for a fixed transaction rate of 15TPS and a batch size of 20.

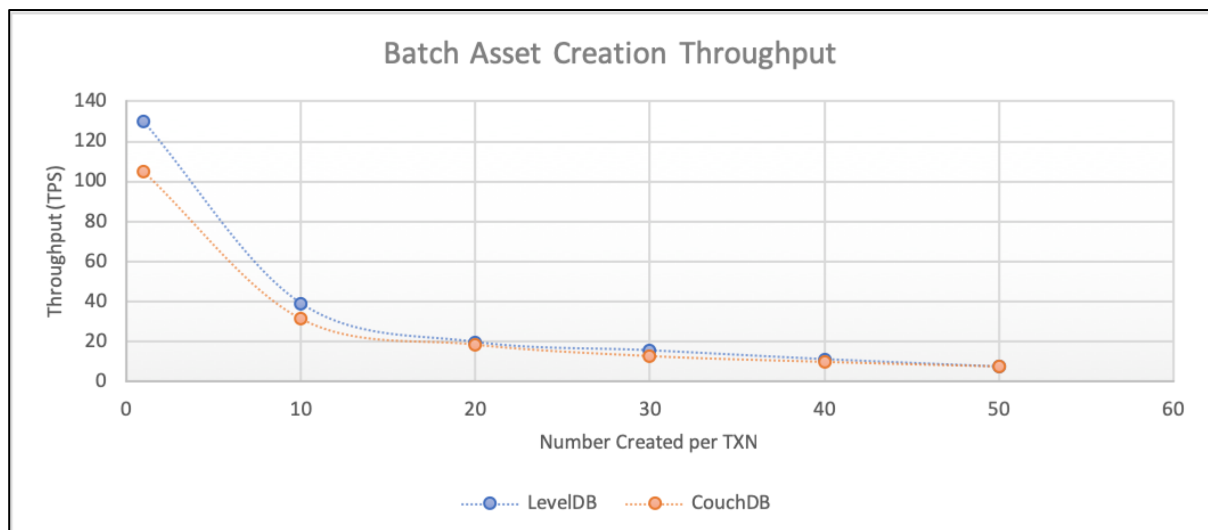
Benchmark Results

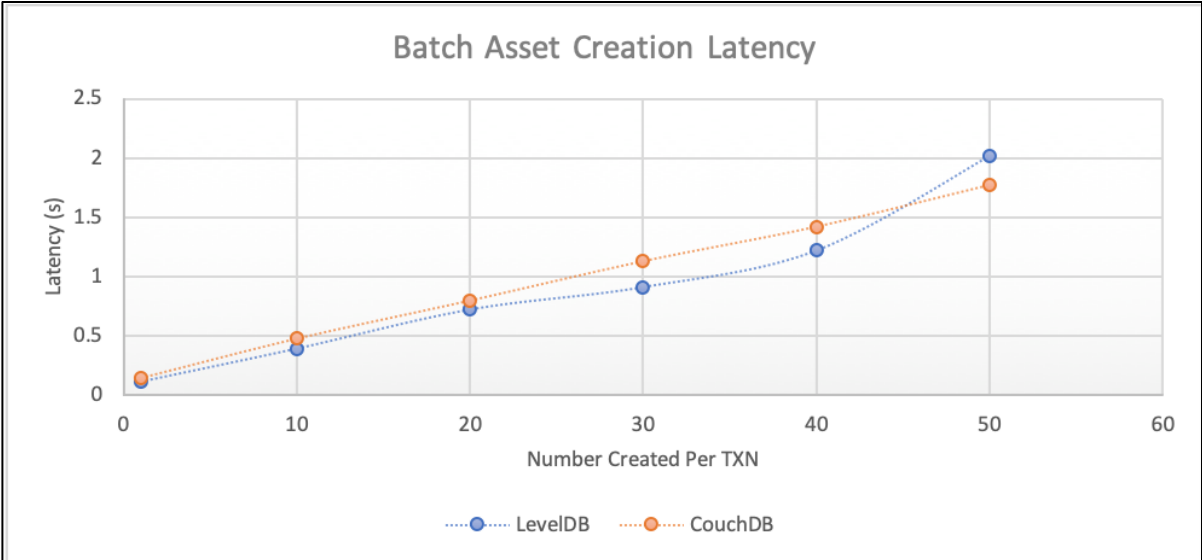
LevelDB

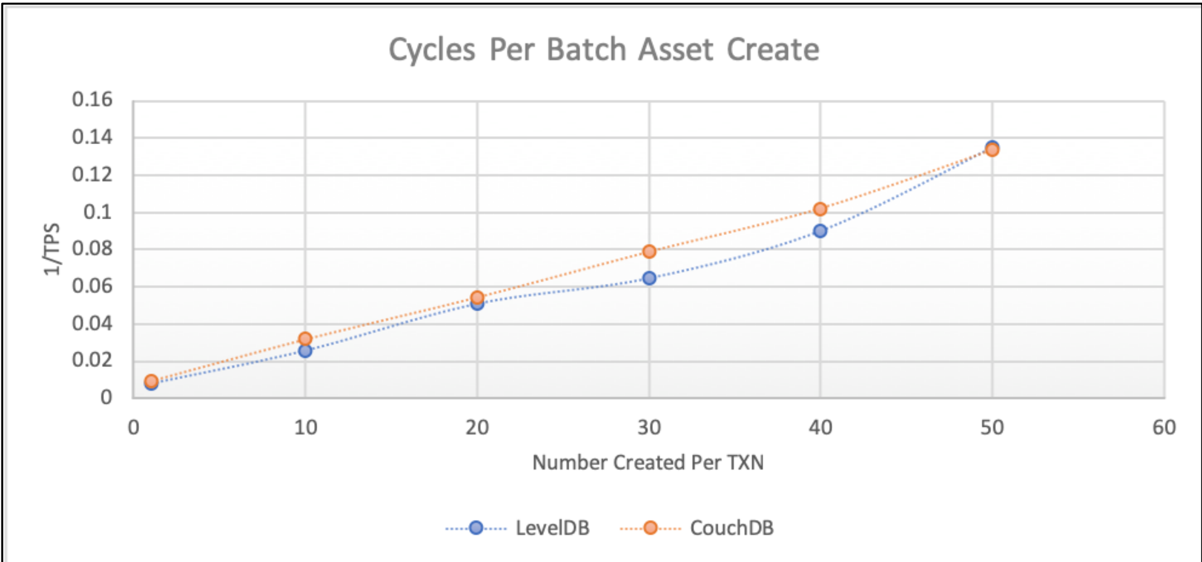
Batch Size	Max Latency (s)	Avg Latency (s)	Throughput (TPS)
1	0.55	0.11	129.8
10	0.85	0.39	39.1
20	2.04	0.72	19.7
30	1.67	0.91	15.5
40	2.39	1.22	11.1
50	8.83	2.02	7.4

CouchDB

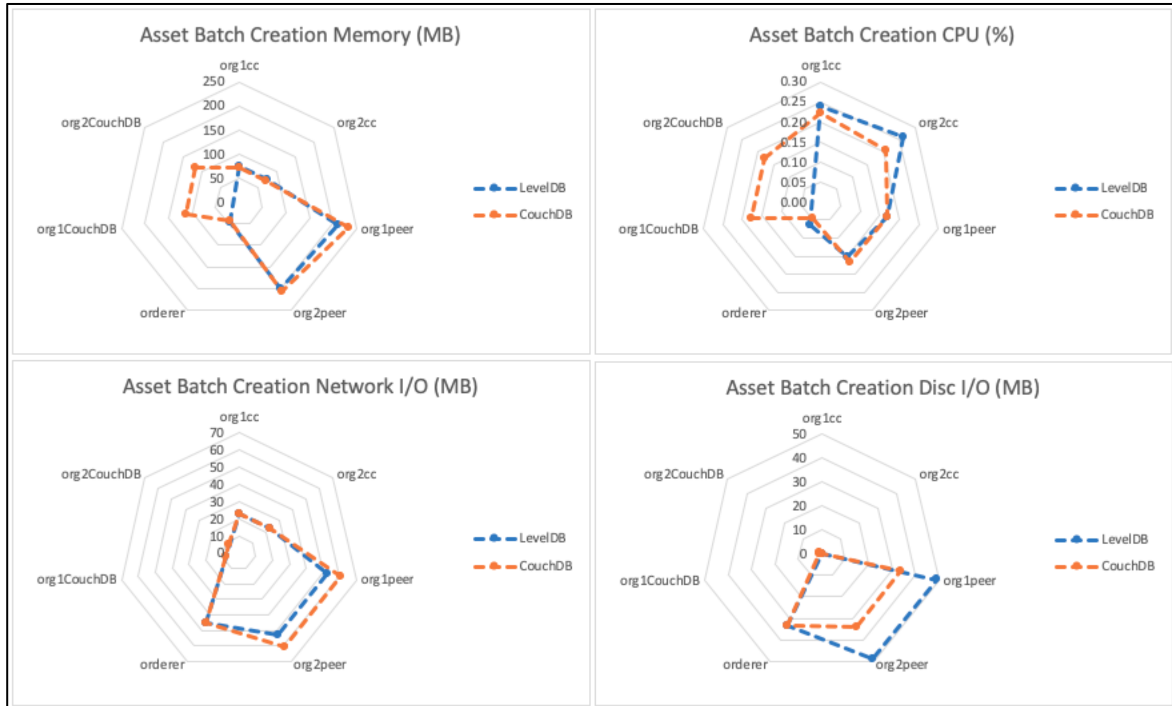
Batch Size	Max Latency (s)	Avg Latency (s)	Throughput (TPS)
1	0.55	0.15	104.9
10	0.93	0.48	31.4
20	1.99	0.80	18.4
30	2.14	1.13	12.7
40	2.82	1.42	9.8
50	3.29	1.77	7.5




Batch Asset Creation Latency
Latency (s)
2.5
2
1.5
1
0.5
0
0
10
20
30
40
50
60
Number Created Per TXN
LevelDB
CouchDB


Cycles Per Batch Asset Create
1/TPS
0.16
0.14
0.12
0.1
0.08
0.06
0.04
0.02
0
0
10
20
30
40
50
60
Number Created Per TXN
LevelDB
CouchDB

Resource Utilization- Batch Size 20 @15TPS



Benchmark Observations

Use of a LevelDB world state database is seen to enable higher throughput and lower latencies with small batch sizes, though this benefit is lost with large batch sizes.

In comparing the resource utilization of a LevelDB world state database with a CouchDB equivalent during batch asset creation, there are similarities with the `Create Asset Benchmark`: implementing a CouchDB world state is CPU intensive, but is observed to be beneficial in terms of disc I/O.

Appendix

Machine Configuration

This report was generated using the following Hyperledger Fabric component levels

- Fabric images: 1.4.0
- Fabric chaincode:1.4.0
- Fabric SDK: 1.4.0

Hyperledger Caliper at commit level 4156c4da7105fd1c2b848573a9943bfc9900becb was used.

The report was generated on an IBM Cloud Softlayer machine with the following configuration:

- OS: Ubuntu 16.04-64
- RAM: 2x16GB Micron 16GB DDR4 2Rx8
- Processor: 3.8GHz Intel Xeon-KabyLake (E3-1270-V6-Quadcore)
- Motherboard: Lenovo Systemx3250-M6
- Firmware: M3E124G 2.10 10-12-2017
- Network Card: Silicom PE310G4i40-T
- HDD: 960GB SanDisk CloudSpeed 1000 SSD
- Architecture: x86_64
- CPU op-mode(s): 32-bit, 64-bit
- Byte Order: Little Endian
- CPU(s): 8
- On-line CPU(s) list: 0-7
- Thread(s) per core: 2
- Core(s) per socket: 4
- Socket(s): 1

Tools

The benchmarking was performed using Hyperledger Caliper, a performance benchmark harness for Hyperledger blockchain solutions. The documentation for the tool contains information and examples on running benchmarks, as well as information in configuring the tool to run the benchmarks used within, and available from, this report.

Resources

The smart contract and Hyperledger Caliper configuration files used in the creation this report are available for download from the caliper-benchmarks GitHub repository.