White paper

Data Bench: A new proof-of-concept workload for microservice transactions

Data Bench is a new proof-of-concept workload that can be used to measure the response-time latency of microservice transactions. This is a new type of workload that places the focus and analysis of computing environments on the handling, processing, and movement of data. Currently in Phase 1 proof-of-concept, this open-source implementation delivers all the required components for an out-of-the-box workload, in a format that is easy to download and easy to use. Data Bench provides a solid starting point for testing the interoperability and delivery mechanisms of future transaction benchmarks for microservices.

AUTHORS

John Fowler
Software and Services Group
Intel Corporation

Erik O'Shaughnessy
Software and Services Group
Intel Corporation

Roy Moore
Software and Services Group
Intel Corporation

Marcus Heckel
Software and Services Group
Intel Corporation

Yingqi Lu
Software and Services Group
Intel Corporation

Jeff Garelick
Software and Services Group
Intel Corporation

INTRODUCTION

There are many benchmarks today that are used to measure various aspects of microservices. Some measure the performance of microservices, some measure Apache Kafka* publish/subscribe aspects, some analyze elements of data storage or data movement. However, none of these benchmarks focus on end-to-end transactions, or allow you to vary the implementation.

Here, we introduce a new open-source workload: Data Bench. Data Bench is designed to help you tune, optimize, develop, and evaluate data-centric computing environments. This workload places the focus and analysis on the handling, processing and the movement of data. Specifically, Data Bench measures the response-time latency for two transactions using Kafka, Apache Spark*, and Apache Cassandra*. Note that other benchmarks use Twitter*, click-stream, or other unstructured data types for big data processing. Unlike those benchmarks, Data Bench uses contemporary online transaction processing (OLTP) structured data for the transactions and the data stored in Cassandra.

Data Bench represents the value and importance placed upon data by companies, customers, data centers, and technology developers. It addresses a critical need for developers who work in microservices. Currently in Phase 1 (proof of concept, or POC), this open-source workload offers an opportunity for the developer community to collaborate in order to further develop a useful method to analyze end-to-end transactions, especially for different microservice implementations.

Data Bench:
A new open source workload

Data Bench is a new, open-source workload for measuring the response-time latency of microservice transactions. This new workload places the focus and analysis of computing environments on the handling, processing, and movement of data.

The workload consists of:

- A new data generator for scalable, flexible, and realistic data for two transactions.
- Scripts for generating tables and for loading from flat files into Apache Cassandra*.
- A fixed dataset size of a couple hundred MB. When developed further, Data Bench will provide the ability to create datasets of any size using a relational schema.
- A streaming transaction of market ticker data called Market-Stream
- An interactive transaction with moderate I/O loads and processing, named Customer-Valuation.
- A driver or transaction generator for generating input data and timing.
- A group of apache Spark* services and microservices to implement the transactions.
- Necessary Apache Kafka* producers and consumers.
- Docker* containers for all of the above.

This initial POC version of Data Bench uses Kafka, Spark, and Cassandra. However, Data Bench can be customized for other environments and configurations.

DATA BENCH PHASE 1

The Phase 1 proof-of-concept Data Bench workload has an architecture based on microservices. This makes it more useful for development and testing in a container-based and scale-out environment.

Data Bench provides a complete implementation of a simple workload. To provide this complete implementation, Data Bench uses several distributed processing applications. Data Bench also uses the latest techniques in publish/subscribe messaging (Kafka), services, microservices (Apache Spark*), and data management (Apache Cassandra*). Data Bench uses Docker* containers to deliver the workload environment efficiently, consistently, and with minimal additional configuration required.

Use of Docker* containers

The use of containers is important in this workload because distributed processing involves replication of the compute environment. It also requires the ability to adjust to demands and interruptions of processing. Any workload for microservices must be able to deliver these ready-to-go environments quickly, accurately, and affordably.

New data model

Data Bench introduces a new data model and new transaction definitions for transactional processing. Data Bench also provides all necessary data, scripts, files, and configurations to help you test your microservice application using this workload.

This new workload incorporates testing and integration of Kafka, Spark, and Cassandra for the management and processing of data and transactions. A Docker container includes a simple, yet effective driver to manage the initiation, pacing, and timing of the individual transactions.

Phase 1 implementation

The phase 1 POC implementation of Data Bench delivers all components in a format that is easy to download and easy to use. The implementation includes a link to several Docker file containers, one for each of the main components: the driver, Kafka, Spark, and Cassandra.

This Phase 1 release also includes instructions for preloading the Cassandra database, along with the Spark services and Kafka topics. When Data Bench is more developed, you should be able to execute the entire workload right out of the box.

WORKLOAD DESIGN

The Data Bench workload is designed to vary the size of the dataset, the percentage mix of individual transactions, and the pacing of incoming transactions. For example, Data Bench provides a mix of heavy, medium, and lightweight transactions. It also provides for periods of average and above-average throughput.

The Data Bench workload is designed to provide and/or include:

- Access the data already configured and stored in Cassandra.
- Transactions to use the functionality of Spark, in order to access and write data to Cassandra, and distribute transactions using Kafka.
- A computing environment and the implementation in Docker containers for separate stages of processing in the workload. This is from transaction generation (driver); to messaging (Kafka); transaction processing (Spark); and data storage and retrieval (Cassandra). The computing environment is implemented in Docker containers for each environment.
- A small, but realistic, fixed-size dataset that is implemented and stored in a Cassandra cluster using a simple and relevant schema.

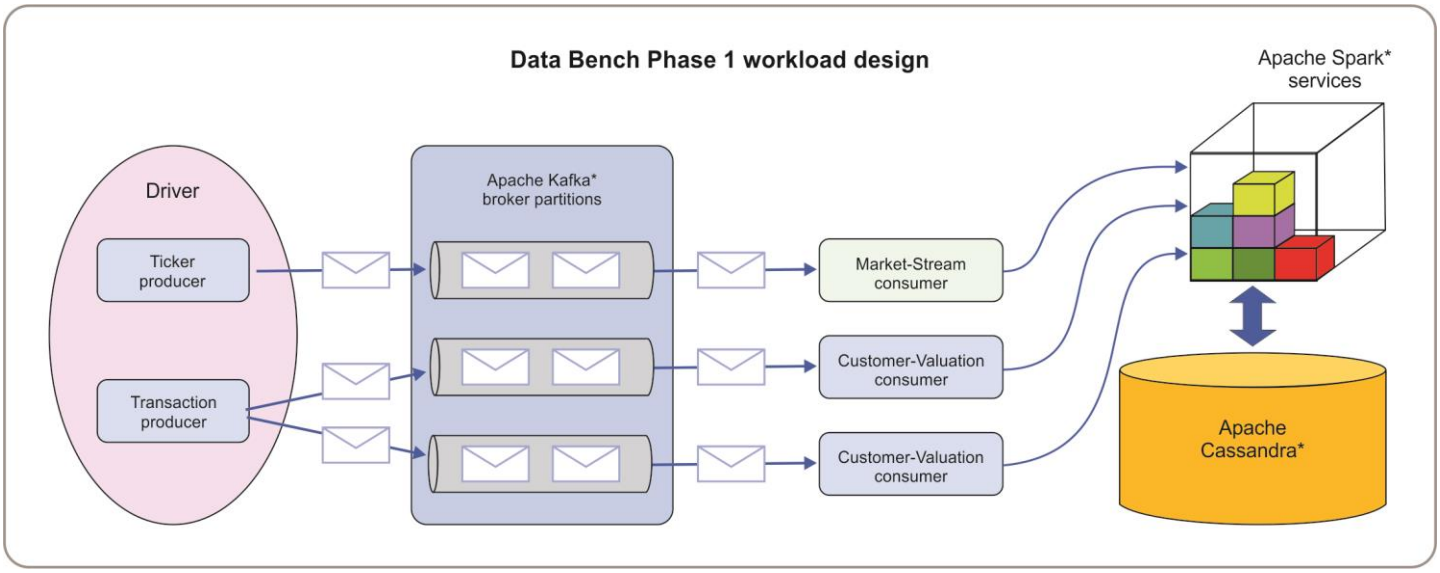

Figure 1. Design for Phase 1 of the new Data Bench workload.

- Two transactions exercise different and challenging aspects of the Kafka-Spark-Cassandra implementation.
- A representation of a market "ticker," to provide a steady stream of input data as a Kafka producer to a cluster of Kafka consumers. The ticker is designed to represent an environment with a steady stream of automatic or non-iterative data collectors (such as sensors and Internet-of-Things devices).
- An interactive transaction that retrieves the ever-changing valuation of a customer's portfolio, and returns that information to the requester.
- A transaction generator (driver) as a Kafka producer that generates the input data for each transaction, and sends the requests through Kafka to the Spark consumers.
- A group of Spark services fronting a Cassandra data manager for retrieving and processing the defined workload.
- The necessary Kafka producers and consumers to return any results of executing the prescribed workload transactions.

DATA BENCH COMPONENTS

The Data Bench proof-of-concept has specific requirements, skills, and steps that make it easier for you to implement. This should help encourage you to try out the workload and the underlying technology without getting bogged down in the configuration. With Data Bench, Intel® provides all components necessary to execute this workload. See Figure 1 (above) for an overview of the Phase 1 Data Bench workload design.

Data generator

Data Bench uses a new flexible, scalable, representative, and freely available input data generator both for the transactions and for populating the tables.

The input data generator creates a reasonably sized dataset. This data set is based on artificial customer and market financial data, in order to represent a workload on the Kafka, Spark, and Cassandra cluster.

Having realistic, scalable, and interdependent data is a vital element to any artificial transactional workload. The data is crucial in order for the workload to better emulate and represent compute environments. However, access to real data from existing environments or customers is almost impossible to acquire. That kind of data cannot be distributed, and it is tightly coupled to an implementation and/or software environment.

For those and other reasons, Data Bench uses the Transaction Processing Performance Council's (TPC) TPC Benchmark* E data generator to create the text files that populate the tables in Cassandra. Data Bench uses only the output of the TPC's code. No TPC code is used during the generation of transaction input data or during transaction execution.

After the text files are generated, the CUSTOMER and CUSTOMER_ACCOUNT tables are flattened and joined together to form a new CUSTOMER_ACCOUNT table. It's that data which is used in this workload.

The Data Bench component that generates the input data required for every transaction, also uses the same text files needed to populate the tables. The text file is read into memory and randomly selects which security symbol, customer identification (ID), or customer tax ID is used for any transaction.

Driver

As shown in Figure 1 (previous page), Data Bench includes a workload driver as part of the environment. The driver is a single system, written in Python Software Foundation Python* with Linux Foundation Kubernetes* command and control. There are separate containers for the customer generator and Market-Stream generator.

The driver provides:

- Input generation for the transactions
- Pacing of transactions
- Collection and tracking of response-time latencies per transaction
- Generation of cumulative performance reports, based on transaction throughput

The driver's processes use Kafka APIs (application programming interfaces) to send and receive messages with the Kafka brokers and topics. The driver is packaged in Docker file containers, and comes preloaded, preconfigured, and ready to run.

Spark services used to manage two transactions

The Spark services manage the execution of the transactions. In this POC, the workload defines and measures the response-time latency for two transactions: Market-Stream and Customer-Valuation. The workload provides a mix of heavy, medium, and lightweight transactions for both Market-Stream and Customer-Valuation.

Data Bench transactions: Market-Stream and Customer-Valuation

The Data Bench workload implements a series of transactions on a small cluster. The workload distributes and balances the load of the individual transactions: heavy, medium, and lightweight; and provides periods of average and above-average throughput. The new data generator in the workload is used to populate the tables.

Market-Stream transaction overview:

- A streaming transaction that represents a continually changing security ticker
- A constant stream of updates to Apache Cassandra* tables, as pricing changes come from the Market Generator (you can customize the rate of updates)
- Future updates to tables will trigger other transactions and/or actions on the data
- The transaction is representative of the environment, with continuous input arriving at constant or variable rates

Customer-Valuation transaction overview:

- Retrieves a customer's profile, and summarizes the overall standing for each account, based on current market values. For every account, this transaction returns:
  - Cash balance and value
  - Quantity for each security
  - Gain and/or loss from the purchase price as compared to current market prices
- The value of the portfolio constantly changes as the market itself changes

Transaction: Market-Stream

The Market-Stream transaction represents a continually changing security ticker. In other words, it is a constant stream of ticker updates coming from the generator. The data stream includes individual symbols, a unique ID, a price quote, and the quantity traded.

In this Phase 1 workload, the generator selects a security symbol, and generates a new price and a quantity. The generator then requests a unique identifier for the transaction, and submits this to the Kafka topic for Market-Stream via the producer API.

A Spark service consumer then retrieves the entry from the Kafka topic. The service processes the ticker symbol by updating the appropriate fields in the Cassandra LAST_TRADE table, and by inserting a new row into the MARKET_STREAM_TXN.

- The LAST_TRADE table is the definitive source of the last price quoted and traded for any security in the system.
- The MARKET_STREAM_TXN table is used to track each of the individual ticker updates made, and then determine the latency of the individual transactions.

In this POC, Data Bench uses a steady stream of data (variable controlled submission rate) with an initial rate of 20 tickers per second. A ticker consists of information from the driver, as described in Table 1 (next page).

In this Phase 1 workload, the Spark service retrieves the incoming structure, and acts on the Cassandra tables, as described in Table 2 (next page).

Transaction: Customer-Valuation

The Customer-Valuation transaction is interactive. In other words, the data is returned to the originator of the transaction, and is timed by the driver. Currently, in this POC, the transaction computes a customer's overall value for:

- All of the securities held
- Cash balance for each account
- Value for each account
- Total cost for the security in each account
- Value for each security in an account based on the last trade price of the security.

The transaction also selects all of the columns from the CUSTOMER_ACCOUNT table based on either the *customer_id* or the *customer_tax_id* from the input.

Data Bench retrieves the security symbols, purchase price, and total quantity matching the customer's account IDs from the HOLDING table. The last trade price is read from the LAST_TRADE table for each security and returned.

The input for the transaction is typically the *customer_id* (for the customer) 70% of the time; or the *customer_tax_id* 30% of the time. If one field is populated, the other must be 0 (zero).

Table 3 describes the Customer-Valuation fields.

Table 1. Ticker information from the driver

Field name	Field type	Description
Transaction Name	String	Set to MarketStream
UUID	Unique ID	System-generated unique string identifier
MST_DTS	Date / timestamp	Date/time when submitted
MST_TXN_CNTR	Integer	Transaction counter
LT_PRICE	Decimal	Price of the trade
LT_QTY	Integer	Number of securities traded
LT_S_SYMB	String	Security symbol traded

Table 2. Spark service actions on Apache Cassandra* tables

Table name	Column	Action
LAST_TRADE	LT_S_SYMB	Access
	LT_PRICE	Update
	LT_VOL	Update
	LT_DTS	Update
MARKET_STREAM_TXN	MST_ID	Insert
	MST_START_DTS	Insert
	MST_END_DTS	Insert
	MST_S_SYMB	Insert
	MST_PRICE	Insert
	MST_QTY	Insert

Table 3. Customer-Valuation transaction fields

Field name	Field type	Description
Transaction Name	String	Set to CustomerValuation
UUID	Unique ID	System-generated unique string identifier
MST_DTS	Date / timestamp	Date/time when submitted
MST_TXN_CNTR	Integer	Transaction counter
customer_id	Int64	Customer ID
customer_tax_id	Int64	Customer Tax ID

Table 4. Operations performed on Cassandra tables by the Customer-Valuation service

Table name	Column	Action
CUSTOMER_ACCOUNT	CA_ID	Return
	CA_C_ID	Compare
	CA_BAL	Return
	CA_F_NAME	Return
	CA_L_NAME	Return
	CA_M_NAME	Return
	CA_TAX_ID	Return
HOLDING	H_CA_ID	Compare
	H_S_SYMB	Return
	H_PRICE	Return
	H_QTY	Return
LAST_TRADE	LT_S_SYMB	Compare
	LT_PRICE	Return

Table 5. Output structure

Field name	Field type	Description
Transaction Name	String	Set to CustomerValuation
UUID	Unique ID	System-generated unique string identifier
UUID_reply	Unique ID	Unique ID of the response returned
txn_sequence	Int	Sequence number for transaction
asset_total[acct_id]	Double	Array of totals, 1 per account
cash_bal[acct_id]	Double	Array of cash balance, 1 per account
acct_id[acct_id]	Int64	Array of account ID, for customer ID
symbol[max][acct_id]*	String	Arrays of security symbols per account, for each account ID
h_qty[max][acct_id]	Int32	Arrays of security quantities per account, for each account ID
h_cost[max][acct_id]	Double	Arrays of security cost per security symbol, for each account ID
h_val[max][acct_id]	Double	Arrays of current security value per security symbol, for each account ID
acct_name[acct_id]	String	Array of customer account names
customer_id	Int64	Customer ID
customer_acct_id[acct_id]	Int64	Array of customer account IDs
first_name	String	Customer first name
middle_name	Char	Customer middle initial
last_name	String	Customer last name

* *For symbol[max][acct_id], max = 10. This is the maximum number of securities per account*

Input structure

The input structure is passed to the driver process/procedure that acts as the Kafka producer. The producer marks the beginning timestamp and a unique transaction identifier for the transaction, and logs for processing later. The driver then produces the transactions as a message to a Kafka topic for processing.

CA_C_ID compare action

Data Bench identifies the CA_C_ID compare action by either the *customer_id* or the *customer_tax_id* from the input structure. If the *customer_tax_id* is provided, then the CA_C_ID is returned.

All of the rows from the CUSTOMER_ACCOUNT table are retrieved for the CA_C_ID.

Next, the rows from HOLDING for the individual CA_IDs are returned. Data Bench uses these to compute:

- Customer's overall value
- Account's current value
- Purchase value for each security held by the customer

LT_PRICE

Data Bench retrieves the LT_PRICE for each security symbol in the customer's holdings from the LAST_TRADE table. This is used to compute:

- Current value of each security
- Total value for each account
- Current total value for the customer

In turn, the workload uses that information to populate the output structure, with the fields described in Table 5.

The driver retrieves the output structure from the Customer-Valuation Spark service via the Kafka consumer APIs. The transaction is now complete.

Cassandra

For this Phase 1 workload, the data manager is Cassandra. Data Bench uses a Docker file container, for which Cassandra is already configured and loaded with the base number of rows.

Tables 6, 7, 8, and 9 (next page) describe the schemas used by the Cassandra tables for CUSTOMER_ACCOUNT, HOLDING, LAST_TRADE, and MARKET_STREAM_TRANSACTION.

Table 6. Schema for CUSTOMER_ACCOUNT

Column name	Column type	Description
CA_C_ID	Int64	Customer ID
CA_ID	Int64	Customer Account ID
CA_TAX_ID	Text	Customer Tax ID
CA_B_ID	Int64	ID of Broker for Customer Account
CA_NAME	Text	Name of the Customer Account
CA_BAL	Decimal	Cash balance for the account
CA_L_NAME	Text	Last name of the customer
CA_F_NAME	Text	First name of the customer
CA_M_NAME	Text	Middle initial of the customer
Primary key: CA_C_ID, CA_ID		

Table 7. Schema for HOLDING

Field name	Field type	Description
H_T_ID	Int64	Trade ID associated with security
H_CA_ID	Int64	Customer Account for security
H_S_SYMB	Text	Security Symbol
H_DTS	Date Timestamp	Date / Time of trade
H_PRICE	Decimal	Value of security at trade
H_QTY	Int32	Number of security with trade
Primary key: H_T_ID, H_CA_ID		

Table 8. Schema for LAST_TRADE

Field name	Field type	Description
LT_S_SYMB	Text	Security symbol
LT_DTS	Date Timestamp	Date/time of last update to symbol
LT_OPEN_PRICE	Decimal	Value of symbol at open of exchange
LT_VOL	Int64	Total number of shares since open
Primary key: LT_S_SYMB		

Table 9. Schema for MARKET_STREAM_TRANSACTION

Field Name	Field Type	Description
MST_ID	Int64	Unique ID of the Market-Stream transaction
MST_START_DTS	Date Timestamp	Date/Time of the start of transaction
MST_END_DTS	Date Timestamp	Date/Time of the end of transaction
MST_S_SYMB	Text	Security symbol
MST_PRICE	Decimal	Price of symbol
MST_QTY	Int32	Number of shares in transaction
Primary key: MST_ID, MST_S_SYMB		

SUMMARY

Data Bench is a new, open-source, data-centric workload that focuses on and analyses the handling, processing, and movement of data. It provides a mix of heavy, medium, and lightweight transactions, as well as periods of average and above-average throughput.

Data Bench is designed to help you tune, optimize, develop, and evaluate your microservices computing environment. With no comparable workload in industry that can handle the movement and storage of microservices data, Data Bench fills a critical need.

Even better, Data Bench is not just a workload that measures the response-time latency of microservice transactions. It also has the hooks necessary to take measurements at various stages in the transaction flow. In turn, this makes it easier for you to find bottlenecks and other areas that could benefit from additional work.

This Phase 1, proof-of-concept implementation of Data Bench delivers all the required components of a benchmark in a format that is easy to download and easy to use.

Right now, Data Bench is a fairly simple workload and, as a proof-of-concept, is not yet well-tuned or stressful. Rather, this Phase 1 Data Bench provides a solid starting point for testing the Kafka-Spark-Cassandra interoperability and delivery mechanism for future benchmarks.

To learn more about Data Bench and how to contribute to the workload, visit the Data Bench repository: **https://github.com/Data-Bench/data-bench**

Printed in USA XXXX/XXX/XXX/XX/XX Please Recycle XXXXXX-001US