redislabs
home of redis

whitepaper

Faster Operational Analytics on Spark Time-Series with Redis

Authors: Sun He, Itamar Haber, Yoav Steinberg, Dvir Volk, Yiftach Shoolman

Table of Contents

Executive Summary

This paper presents results of research done by Redis Labs on accelerating real-time analytics of operational data using in-memory computing. This part of our research focuses on a mainstream operational analytics use case, that of time series data, to demonstrate how it can be optimized with the combination of two best-of-breed solutions: Redis and Apache Spark. Apache Spark is designed for performance but by leveraging on Redis' in-memory nature and powerful data structures we were able to dramatically accelerate the performance of queries that are typical to time-series analysis. As shown by the results, these performance gains allow for anywhere between 20-140 times faster execution times compared to existing implementations.

Introduction

The explosion in data, a.k.a. "Big Data", generated by web applications, financial services, mobile clients and the IoT is only surpassed by the need for processing it in near real-time and extracting meaningful insights. The last decade has been an exciting and tumultuous one that gave birth to new models and approaches dedicated to satisfying the need for operational analytics to solve business questions. In-memory computing has always been the keystone for delivering results in a timely manner and as time moves on we discover ways to combine proven solutions and novel approaches to address new challenges better.

What is time-series data

Data comes in many shapes and sizes but a large part of the data that we create it is irrevocably linked to time. Because we are three-dimensional beings who only move forward in time at constant pace, ordering our observations by the only real static vector is not only natural but also allows us to make educated predications about the future.

The term time-series is therefore used to describe data, the measurement of some value, that is ordered by time. In this paper, we use randomly-generated financial data (stock prices) for input. However, the benchmark equally applies to other types of time-series data including sensor readings, statistical data, econometric data, human- and machine-generated events or any other measurements taken over periods of time.

What is Redis

Redis is an open source, in-memory Data Structure Store, used as a database, a caching layer or a message broker. Sometimes referred to as the "Leatherman of Databases", its simple yet flexible design philosophy makes it an effective choice for solving a multitude of demanding data processing tasks. Redis data structures resolve very complex programming problems with simple commands executed within the data store, reducing coding effort, increasing throughput, and reducing latency.

As such, Redis practically lends itself for real-time time-series data management and analysis. A recent InfoQ article, "Using Redis as a Time Series Database: Why and How" from Dr. Josiah Carlson, explains how time-series data can be optimally modeled and queried in Redis by using its Sorted Set and Hash data structures.

What is Apache Spark

The open source Apache Spark project is a new and exciting technology that empowers us to process data in ways that were never possible before. Spark is designed for facilitating the in-memory manipulation of large volumes of data in a scalable manner via cross-cluster parallelization. It is a framework, an engine and a growing number of libraries that make large-scale data processing fast and easy.

What is Spark Time-Series

Spark Time-Series, or Spark-TS, is a library by Cloudera for analyzing time-series data with Apache Spark. The spark-timeseries library is open source and an excellent post from Cloudera's Data Science team who developed it provides a deep dive into its design and usage.

In broad terms, the Spark-TS library allows usage of Spark for reading raw time-series data from Spark's data sources (e.g. regular files from disk and HDFS or data from a database) and manages the indexing data structures that facilitate efficient time range queries (slicing). It allows the data to be processed using both the framework's main abstraction – the time-series resilient distributed dataset, or TimeSeriesRDD – and via the Observations and Instants DataFrames. Spark-TS provides an optimized interface for manipulating the time-series data (alignment, slicing and missing value imputation) as well as useful statistical methods for analyzing it.

Spark Performance

Having been designed for fast and distributed in-memory data processing, Spark is well optimized for performance and includes robust tools that take advantage of its workers' RAM out of the box. Spark worker processes use local memory to store and process the dataset. The structures used for the data are stored in the memory in raw deserialized form, allowing for optimal access times but resulting in memory overheads of between two to four times the size of actual data. On top of consuming a lot of memory, this approach also triggers frequent garbage collection once the memory size reaches a certain threshold, which can dramatically degrade performance.

Spark on-heap cache

One of Spark's most important capabilities is the ability to cache (persist) datasets so they can be reused across multiple operations, thus eliminating the need to reload them from the source. A dataset can be cached on-heap, that is in the memory of the JVM that runs the work process, in either deserialized or serialized form. Because the JVM's heap size is finite, cached datasets may also be configured to be persisted to the worker's local disk when they can't be kept in RAM, to avoid the need to recompute them.

While dataset objects in serialized form require more time be retrieved from the cache due to the processing needed to deserialize them, they have nearly no space overhead and are thus are much smaller. Since recomputing the dataset and accessing disk are costlier compared to the deserialization penalty, caching of serialized objects is an acceptable compromise.

Off-heap cache

To overcome the size limit of on-heap caching and reduce the performance impact of JVM's garbage collector due to cache activity, Spark also provides the means for off-heap caching – that is out of process – of serialized dataset objects. This experimental interface persists the dataset to Tachyon, the memory-centric distributed storage system, and is optimized for moving large amounts data between Spark's workers and Tachyon's processes via ramfs.

Speeding Up Spark-TS

Given Spark's out-of-the-box performance and its existing caching strategies we sought to optimize the common operations with the biggest performance gain. For that purpose we elected filtering by time ranges and slicing operations, with these being the basis for any analytical workflow.

The spark-redis library

The first step we took in accelerating Spark-TS is based on the spark-redis connector library. The connector library allows a Redis database to be used as a data source for Spark, defining a standard way for reading and writing data from Redis' native data structures to RDDs and vice versa. Because Redis is an in-memory database, it delivers the highest data throughput at the lowest possible latencies, making it a perfect choice for supporting the requirements of data-hungry processes.

The spark-redis connector is also Redis-cluster aware, and can thus parallelize RDDs according to the same partitioning scheme that the data source uses. This capability minimizes traffic between the Spark and Redis processes and can also be used for keeping the data close to the CPU on shared infrastructure for additional performance gains.

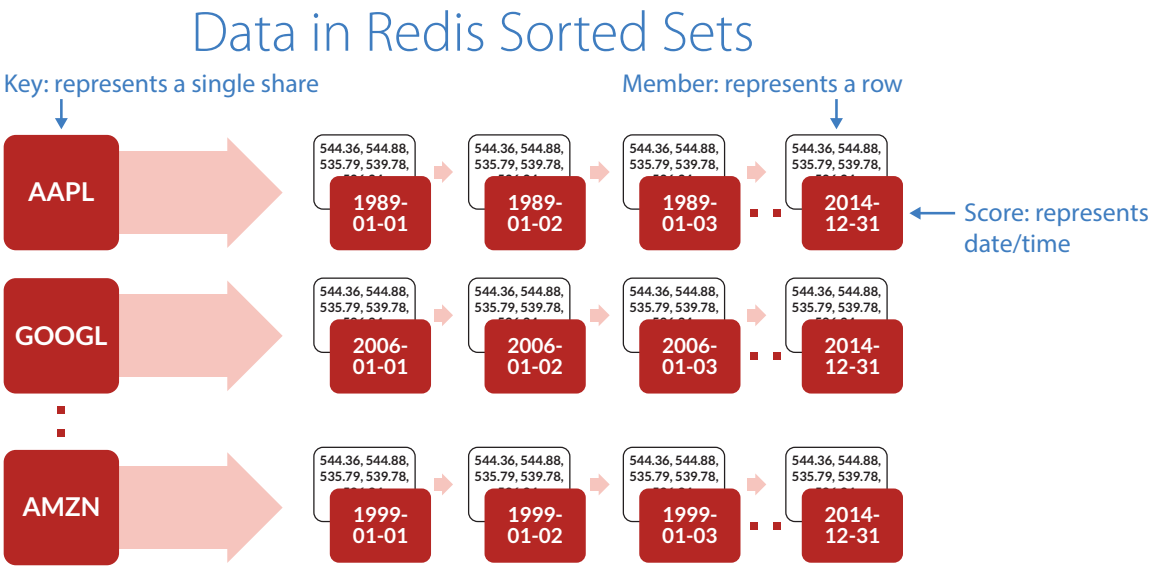
Redis optimized for time-series storage

Just having the time-series stored in Redis results in superior performance compared to that of uncached datasets. In that capacity and since data is served directly from RAM, Redis is essentially an off-heap cache. That alone does not provide any significant gains over Spark's existing caching mechanisms, so the second step we took was to exploit the properties of Redis' Sorted Set data structure for storing the data.

Having the data stored optimally to reply to the queries about it is Redis' real power. For the time-series data, we used a Sorted Set to store each share's time-series. The following is an example of raw time-series for a single share:

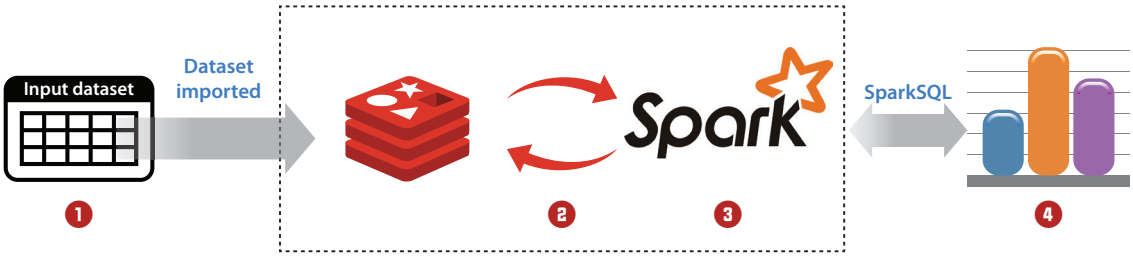
Date	Open	High	Low	Close	Volume	Adj Close
24/10/14	544.36	544.88	535.79	539.78	1967700	539.78
23/10/14	539.32	547.22	535.85	543.98	2342400	543.98
22/10/14	529.89	539.8	528.8	532.71	2911300	532.71

Both the time-stamp and the values of the time-series are stored in the Sorted Set, allowing for efficient range lookups and ordering of the results. Furthermore, the index of time-series is also maintained to facilitate efficient scans. This data model is easily sharded across a Redis cluster that can be scaled to be meet any performance requirements. The diagram below depicts the data's layout in Redis:



RedisTimeSeriesRDD and DataFrames

In order to provide the required level of abstraction of the time-series data's storage model in Redis to Spark's processes, we've created an open fork of the spark-timeseries repository and added to it the RedisTimeSeriesRDD. This RDD can be used in place of the Spark-TS' TimeSeriesRDD with the only difference being that it uses the data in Redis. In addition, Redis' spark-timeseries also implements the Instants and Observations DataFrames, allowing the use of SparkSQL with the data in Redis via the com.redislabs.provider.sql provider.



The block diagram above illustrates the operation principles of spark-timeseries analysis with Redis, identified by the following steps:

1. The dataset is imported into Redis' data structures
2. Redis' data structures are abstracted via the RDD, DataFrame and DataSource APIs
3. The dataset is manipulated with the RedisTimeSeriesRDD, which provides the optimal manner way for querying and persisting it
4. SparkSQL queries can be used on the dataset as well

Testing the Performance of Redis Spark-Timeseries

Performance testing for Spark-TS is implemented with a test suite that compares the four different performance strategies, namely:

1. Uncached - data resides on HDFS, indices are in RAM
2. Tachyon - data is serialized and cached off-heap, indices are in RAM
3. In-process - data and indices are serialized and cached in the process' RAM
4. Redis - access to data is via the RedisTimeSeriesRDD

At the test's beginning the dataset is loaded and then, for each of the strategies, the suite measures the execution time of several variants of time range queries, including filtering by start time, end time, both or regular expressions. Each of the suite's test queries is also run three times, once using the the TimeSeries RDD and twice by using SparkSQL with the Instants and Observations DataFrames.

Dataset

The test uses time-series of randomly-generated financial data. We used the test suite on a dataset consisting of 1024 raw data files, each being the time-series of a single financial share's prices over a 32 year period.

A representative example of the data is here, and the script used for generating the test dataset can be found here.

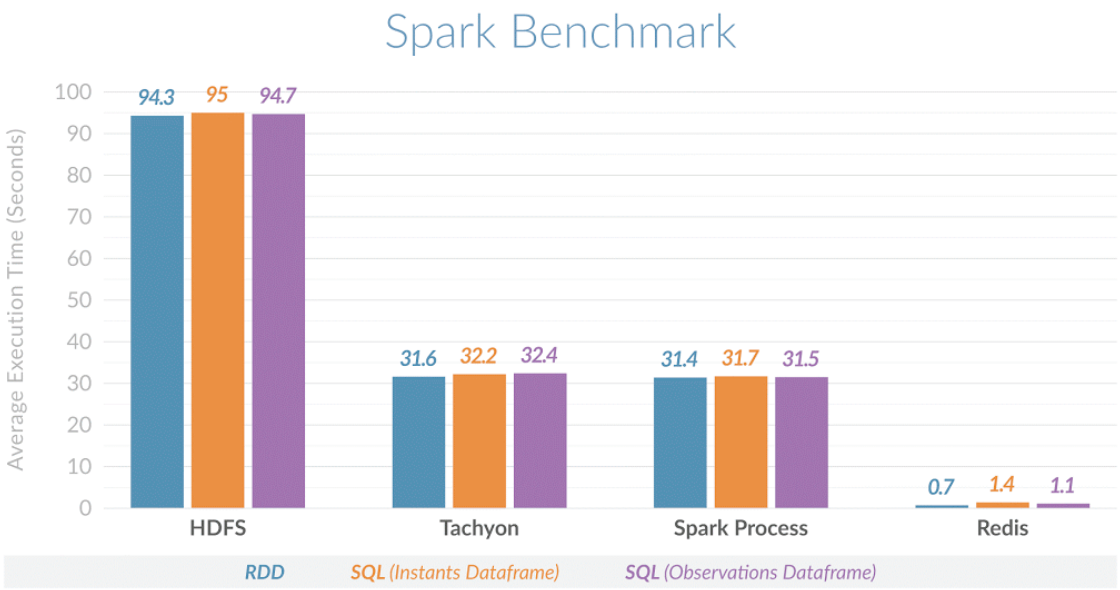
Environment

The tests were conducted on a cluster consisting of five nodes in AWS' N. Virginia data region. Each of the cluster's nodes was an c3.4xlarge EC2 instance with 30GB of RAM and 16 vCPUs. The nodes were installed with the following stack:

- Amazon Linux AMI - kernel 4.1.10
- Apache Spark 1.4.0
- Redis Cluster 3.0.5

Representative Results

Filter by start and end time, slice 1 year



The chart above shows the results obtained from our performance measurements. The example presents the average execution time for a query that performs a range filter and then extracts a single year from it. Each group represents one of the performance strategies and the columns in each group reflect the performance of the library's different abstractions.

It is hardly surprising that the uncached, HDFS-based approach exhibits the worst performance scores. Both Tachyon and Spark's in-process serialized caches provide remarkably similar but constant and significant performance improvements over the uncached results, cutting execution times by as much as 66%. The best performance scores, by orders of magnitude, are obtained from Redis.

The following table summarizes Redis' comparative performance gain in this test case:

Gain in performance with Redis	vs. Uncached	vs. Tachyon	vs. In-process
TimeSeries RDD	134.71 times faster	45.14 times faster	44.86 times faster
SparkSQL Instants	67.86 times faster	23 times faster	22.64 times faster
SparkSQL Observations	86.09 times faster	29.45 times faster	28.64 times faster

Summary

This paper presents an advanced optimization to what is already an extremely efficient data processing flow. While Apache Spark and Cloudera's Spark-TS library are powerful and effective choices for processing time-series in speed and at scale, any gains in performance are critical for real-time operational use cases.

Using Redis as the workflow's time-series source allows cutting down the execution time of typical operations by orders of magnitude. By modeling the data store to the data's properties and the query patterns against the dataset, we were able significantly reduce the network resources needed for data transfer and push part of its filtering to the data store itself.

Apache Spark and the ecosystem around it present many opportunities for using Redis for superior performance. Additional development of spark-redis is planned to include an implementation of Spark's new DataSets AP. We're also researching typical use cases for Spark GraphX and Spark MLlib that can benefit from Redis' capabilities - if you're interested feel free to contact us at expert@redislabs.com.