Azure Cosmos DB performance tips for Async Java | Azure

rockboyfor

[!div class=“op\_single\_selector”] \* [Async Java](performance-tips-async-java.md) \* [Java](performance-tips-java.md) \* [.NET](performance-tips.md)

# Performance tips for Azure Cosmos DB and Async Java

Azure Cosmos DB is a fast and flexible distributed database that scales seamlessly with guaranteed latency and throughput. You do not have to make major architecture changes or write complex code to scale your database with Azure Cosmos DB. Scaling up and down is as easy as making a single API call or SDK method call. However, because Azure Cosmos DB is accessed via network calls there are client-side optimizations you can make to achieve peak performance when using the [SQL Async Java SDK](sql-api-sdk-async-java.md).

So if you’re asking “How can I improve my database performance?” consider the following options:

## Networking

1. **Collocate clients in same Azure region for performance**

When possible, place any applications calling Azure Cosmos DB in the same region as the Azure Cosmos DB database. For an approximate comparison, calls to Azure Cosmos DB within the same region complete within 1-2 ms, but the latency between the West and East coast of the US is >50 ms. This latency can likely vary from request to request depending on the route taken by the request as it passes from the client to the Azure datacenter boundary. The lowest possible latency is achieved by ensuring the calling application is located within the same Azure region as the provisioned Azure Cosmos DB endpoint. For a list of available regions, see [Azure Regions](https://www.azure.cn/support/service-dashboard/#services).  
  
![Illustration of the Azure Cosmos DB connection policy](./media/performance-tips/same-region.png)

## SDK Usage

1. **Install the most recent SDK**

* The Azure Cosmos DB SDKs are constantly being improved to provide the best performance. See the [Azure Cosmos DB SDK](sql-api-sdk-async-java.md) pages to determine the most recent SDK and review improvements.

1. **Use a singleton Azure Cosmos DB client for the lifetime of your application**

* Each AsyncDocumentClient instance is thread-safe and performs efficient connection management and address caching. To allow efficient connection management and better performance by AsyncDocumentClient, it is recommended to use a single instance of AsyncDocumentClient per AppDomain for the lifetime of the application.

1. **Tuning ConnectionPolicy**

* Azure Cosmos DB requests are made over HTTPS/REST when using the Async Java SDK, and are subjected to the default max connection pool size (1000). This default value should be ideal for the majority of use cases. However, in case you have a very large collection with many partitions, you can set the max connection pool size to a larger number (say, 1500) using setMaxPoolSize.

1. **Tuning parallel queries for partitioned collections**

* Azure Cosmos DB SQL Async Java SDK supports parallel queries, which enable you to query a partitioned collection in parallel (see [Working with the SDKs](sql-api-partition-data.md#working-with-the-azure-cosmos-db-sdks) and the related [code samples](https://github.com/Azure/azure-cosmosdb-java/tree/master/examples/src/test/java/com/microsoft/azure/cosmosdb/rx/examples) for more info). Parallel queries are designed to improve query latency and throughput over their serial counterpart.
  1. ***Tuning setMaxDegreeOfParallelism:*** Parallel queries work by querying multiple partitions in parallel. However, data from an individual partitioned collection is fetched serially with respect to the query. So, use setMaxDegreeOfParallelism to set the number of partitions that has the maximum chance of achieving the most performant query, provided all other system conditions remain the same. If you don’t know the number of partitions, you can use setMaxDegreeOfParallelism to set a high number, and the system chooses the minimum (number of partitions, user provided input) as the maximum degree of parallelism.
* It is important to note that parallel queries produce the best benefits if the data is evenly distributed across all partitions with respect to the query. If the partitioned collection is partitioned such a way that all or a majority of the data returned by a query is concentrated in a few partitions (one partition in worst case), then the performance of the query would be bottlenecked by those partitions.
  1. ***Tuning setMaxBufferedItemCount:*** Parallel query is designed to pre-fetch results while the current batch of results is being processed by the client. The pre-fetching helps in overall latency improvement of a query. setMaxBufferedItemCount limits the number of pre-fetched results. Setting setMaxBufferedItemCount to the expected number of results returned (or a higher number) enables the query to receive maximum benefit from pre-fetching.
* Pre-fetching works the same way irrespective of the MaxDegreeOfParallelism, and there is a single buffer for the data from all partitions.

1. **Implement backoff at getRetryAfterInMilliseconds intervals**

* During performance testing, you should increase load until a small rate of requests get throttled. If throttled, the client application should backoff for the server-specified retry interval. Respecting the backoff ensures that you spend minimal amount of time waiting between retries. For more information, see [Exceeding reserved throughput limits](request-units.md#RequestRateTooLarge) and DocumentClientException.getRetryAfterInMilliseconds.

1. **Scale out your client-workload**

* If you are testing at high throughput levels (>50,000 RU/s), the client application may become the bottleneck due to the machine capping out on CPU or network utilization. If you reach this point, you can continue to push the Azure Cosmos DB account further by scaling out your client applications across multiple servers.

1. **Use name based addressing**

* Use name-based addressing, where links have the format dbs/MyDatabaseId/colls/MyCollectionId/docs/MyDocumentId, instead of SelfLinks (\_self), which have the format dbs/<database\_rid>/colls/<collection\_rid>/docs/<document\_rid> to avoid retrieving ResourceIds of all the resources used to construct the link. Also, as these resources get recreated (possibly with same name), caching them may not help.

1. **Tune the page size for queries/read feeds for better performance**

* When performing a bulk read of documents by using read feed functionality (for example, readDocuments) or when issuing a SQL query, the results are returned in a segmented fashion if the result set is too large. By default, results are returned in chunks of 100 items or 1 MB, whichever limit is hit first.
* To reduce the number of network round trips required to retrieve all applicable results, you can increase the page size using the [x-ms-max-item-count](https://docs.microsoft.com/rest/api/cosmos-db/common-cosmosdb-rest-request-headers) request header to up to 1000. In cases where you need to display only a few results, for example, if your user interface or application API returns only 10 results a time, you can also decrease the page size to 10 to reduce the throughput consumed for reads and queries.
* You may also set the page size using the setMaxItemCount method.

1. **Use Appropriate Scheduler (Avoid stealing Eventloop IO Netty threads)**

* The Async Java SDK uses [netty](https://netty.io/) for non-blocking IO. The SDK uses a fixed number of IO netty eventloop threads (as many CPU cores your machine has) for executing IO operations. The Observable returned by API emits the result on one of the shared IO eventloop netty threads. So it is important to not block the shared IO eventloop netty threads. Doing CPU intensive work or blocking operation on the IO eventloop netty thread may cause deadlock or significantly reduce SDK throughput.
* For example the following code executes a cpu intensive work on the eventloop IO netty thread:
* Observable<ResourceResponse<Document>> createDocObs = asyncDocumentClient.createDocument(  
   collectionLink, document, null, true);  
    
  createDocObs.subscribe(  
   resourceResponse -> {  
   //this is executed on eventloop IO netty thread.  
   //the eventloop thread is shared and is meant to return back quickly.  
   //  
   // DON'T do this on eventloop IO netty thread.  
   veryCpuIntensiveWork();  
   });
* After result is received if you want to do CPU intensive work on the result you should avoid doing so on eventloop IO netty thread. You can instead provide your own Scheduler to provide your own thread for running your work.
* import rx.schedulers;  
    
  Observable<ResourceResponse<Document>> createDocObs = asyncDocumentClient.createDocument(  
   collectionLink, document, null, true);  
    
  createDocObs.subscribeOn(Schedulers.computation())  
  subscribe(  
   resourceResponse -> {  
   // this is executed on threads provided by Scheduler.computation()  
   // Schedulers.computation() should be used only when:  
   // 1. The work is cpu intensive   
   // 2. You are not doing blocking IO, thread sleep, etc. in this thread against other resources.  
   veryCpuIntensiveWork();  
   });
* Based on the type of your work you should use the appropriate existing RxJava Scheduler for your work. Read here [Schedulers](http://reactivex.io/RxJava/1.x/javadoc/rx/schedulers/Schedulers.html).
* For More Information, Please look at the [Github page](https://github.com/Azure/azure-cosmosdb-java) for Async Java SDK.

1. **Disable netty’s logging** Netty library logging is chatty and needs to be turned off (suppressing log in the configuration may not be enough) to avoid additional CPU costs. If you are not in debugging mode, disable netty’s logging altogether. So if you are using log4j to remove the additional CPU costs incurred by org.apache.log4j.Category.callAppenders() from netty add the following line to your codebase:

* org.apache.log4j.Logger.getLogger("io.netty").setLevel(org.apache.log4j.Level.OFF);

1. **OS Open files Resource Limit** Some Linux systems (like Redhat) have an upper limit on the number of open files and so the total number of connections. Run the following to view the current limits:

* ulimit -a
* The number of open files (nofile) needs to be large enough to have enough room for your configured connection pool size and other open files by the OS. It can be modified to allow for a larger connection pool size.
* Open the limits.conf file:
* vim /etc/security/limits.conf
* Add/modify the following lines:
* \* - nofile 100000

1. **Use native SSL implementation for netty** Netty can use OpenSSL directly for SSL implementation stack to achieve better performance. In the absence of this configuration netty will fall back to Java’s default SSL implementation.

* on Ubuntu:
* sudo apt-get install openssl  
  sudo apt-get install libapr1
* and add the following dependency to your project maven dependencies:
* <dependency>  
   <groupId>io.netty</groupId>  
   <artifactId>netty-tcnative</artifactId>  
   <version>2.0.7.Final</version>  
   <classifier>linux-x86\_64</classifier>  
  </dependency>

For other platforms (Redhat, Windows, Mac, etc.,) refer to these instructions https://netty.io/wiki/forked-tomcat-native.html

## Indexing Policy

1. **Exclude unused paths from indexing for faster writes**

* Azure Cosmos DB’s indexing policy allows you to specify which document paths to include or exclude from indexing by leveraging Indexing Paths (setIncludedPaths and setExcludedPaths). The use of indexing paths can offer improved write performance and lower index storage for scenarios in which the query patterns are known beforehand, as indexing costs are directly correlated to the number of unique paths indexed. For example, the following code shows how to exclude an entire section of the documents (a.k.a. a subtree) from indexing using the “\*" wildcard.
* Index numberIndex = Index.Range(DataType.Number);  
  numberIndex.set("precision", -1);  
  indexes.add(numberIndex);  
  includedPath.setIndexes(indexes);  
  includedPaths.add(includedPath);  
  indexingPolicy.setIncludedPaths(includedPaths);  
  collectionDefinition.setIndexingPolicy(indexingPolicy);
* For more information, see [Azure Cosmos DB indexing policies](indexing-policies.md).

## Throughput

1. **Measure and tune for lower request units/second usage**

* Azure Cosmos DB offers a rich set of database operations including relational and hierarchical queries with UDFs, stored procedures, and triggers - all operating on the documents within a database collection. The cost associated with each of these operations varies based on the CPU, IO, and memory required to complete the operation. Instead of thinking about and managing hardware resources, you can think of a request unit (RU) as a single measure for the resources required to perform various database operations and service an application request.
* Throughput is provisioned based on the number of [request units](request-units.md) set for each container. Request unit consumption is evaluated as a rate per second. Applications that exceed the provisioned request unit rate for their container are limited until the rate drops below the provisioned level for the container. If your application requires a higher level of throughput, you can increase your throughput by provisioning additional request units.
* The complexity of a query impacts how many request units are consumed for an operation. The number of predicates, nature of the predicates, number of UDFs, and the size of the source data set all influence the cost of query operations.
* To measure the overhead of any operation (create, update, or delete), inspect the [x-ms-request-charge](https://docs.microsoft.com/rest/api/cosmos-db/common-cosmosdb-rest-request-headers) header to measure the number of request units consumed by these operations. You can also look at the equivalent RequestCharge property in ResourceResponse or FeedResponse.
* ResourceResponse<Document> response = asyncClient.createDocument(collectionLink, documentDefinition, null,  
   false).toBlocking.single();  
  response.getRequestCharge();
* The request charge returned in this header is a fraction of your provisioned throughput. For example, if you have 2000 RU/s provisioned, and if the preceding query returns 1000 1KB-documents, the cost of the operation is 1000. As such, within one second, the server honors only two such requests before throttling subsequent requests. For more information, see [Request units](request-units.md) and the [request unit calculator](https://www.documentdb.com/capacityplanner).

1. **Handle rate limiting/request rate too large**

* When a client attempts to exceed the reserved throughput for an account, there is no performance degradation at the server and no use of throughput capacity beyond the reserved level. The server will preemptively end the request with RequestRateTooLarge (HTTP status code 429) and return the [x-ms-retry-after-ms](https://docs.microsoft.com/rest/api/cosmos-db/common-cosmosdb-rest-request-headers) header indicating the amount of time, in milliseconds, that the user must wait before reattempting the request.
* HTTP Status 429,  
   Status Line: RequestRateTooLarge  
   x-ms-retry-after-ms :100
* The SDKs all implicitly catch this response, respect the server-specified retry-after header, and retry the request. Unless your account is being accessed concurrently by multiple clients, the next retry will succeed.
* If you have more than one client cumulatively operating consistently above the request rate, the default retry count currently set to 9 internally by the client may not suffice; in this case, the client throws a DocumentClientException with status code 429 to the application. The default retry count can be changed by using setRetryOptions on the ConnectionPolicy instance. By default, the DocumentClientException with status code 429 is returned after a cumulative wait time of 30 seconds if the request continues to operate above the request rate. This occurs even when the current retry count is less than the max retry count, be it the default of 9 or a user-defined value.
* While the automated retry behavior helps to improve resiliency and usability for the most applications, it might come at odds when doing performance benchmarks, especially when measuring latency. The client-observed latency will spike if the experiment hits the server throttle and causes the client SDK to silently retry. To avoid latency spikes during performance experiments, measure the charge returned by each operation and ensure that requests are operating below the reserved request rate. For more information, see [Request units](request-units.md).

1. **Design for smaller documents for higher throughput**

* The request charge (the request processing cost) of a given operation is directly correlated to the size of the document. Operations on large documents cost more than operations for small documents.

## Next steps

To learn more about designing your application for scale and high performance, see [Partitioning and scaling in Azure Cosmos DB](partition-data.md).