Index-Selection for Minimizing Costs of a NoSQL Cloud Database

Sudarshan S. Chawathe[0000−0001−6930−0325]

University of Maine, Orono ME 04469, USA
chaw@eip10.org
http://chaw.eip10.org/

Abstract. The index-selection problem in database systems is that of determining a set of indexes (data-access paths) that minimizes the costs of database operations. Although this problem has received significant attention in the context of relational database systems, the established methods and tools do not translate easily to the context of modern non-relational database systems (so-called NoSQL systems) that are widely used in cloud and grid computing, and in particular systems such as DynamoDB from Amazon Web Services. Although the index-selection problem in these contexts appears simple at first glance, due to the very limited indexing features, this simplicity is deceptive because the non-relational nature of these databases and indexes permits more complex indexing schemes to be expressed. This paper motivates and describes the index-selection problem for NoSQL databases, and DynamoDB in particular. It motivates and outlines a cost model to capture the specific monetary costs associated with database operations in this context. The cost model has not only been carefully checked for consistency using the system documentation but also been verified using actual usage costs in a live DynamoDB instance.

Keywords: Cloud Computing · Cost Model · Index Selection · DynamoDB · NoSQL Databases · Physical Database Design.

1 Introduction

Recent years have witnessed a rapid growth in usage of diverse *database and storage services in cloud-based systems* such as *Amazon Web Services (AWS)* and *Microsoft Azure*. This growth is fueled by several attractive features of these systems, chief among which are their low start-up costs and their ability to scale rapidly (in a few minutes) over several orders of magnitude of throughput and other system metrics. Related advantages include resilience to a variety of system failures and the global reach of their network of data-centers. Such services often use a nonrelational, or so-called *NoSQL*, data model which permits more flexibility but also complicates database design.

An important feature of these database services is the method used for *billing for usage*. In contrast to the static or slowly changing billing methods used by

conventional on-site relational database systems, cloud-based NoSQL systems typically use billing methods that are sensitive to changes in provisioning and use at time scales of minutes or hours rather than weeks or months. In addition to the finer temporal granularity of billing, these systems also tend to employ finer billing granularity in details of usage, based on a variety of measured and billed metrics. While effective use of such a fine-grained cost structure is potentially advantageous to both client and service provider, and potentially more efficient in a financial sense, realizing such effective use is often very challenging due to the complexity of the cost and billing model.

Secondary indexes, i.e., alternate efficient access paths to data stored in a table, are a crucial feature of database systems, both relational and nonrelational, that permit efficient use of system resources. An important aspect of cloud database services is that system resources are off-site with the service provider and the use of those resources is mapped to monetary costs at a finer granularity. In any database (relational or NoSQL) of even moderate complexity, the *index selection* task, of determining which secondary indexes should be created (and maintained), is a difficult one. The number of possibly useful indexes grows very rapidly with the size of the database schema (table structure), rendering exhaustive examination of all possibilities impracticable. Furthermore, even the conceptually simpler task of determining the degree of merit of a given scheme of secondary indexes is a difficult one, albeit for differing reasons in conventional hosted relational databases and cloud-based NoSQL databases. In the former, a chief source of difficulty is determining how a typically sophisticated but necessarily non-optimal (due to the combinatorics involved) query optimizer will respond to a given scheme of indexes. Recent advances in that context have favored probing the optimizer to answer this question over attempting to model its complex behavior independently. In contrast, NoSQL systems are typically characterized by very simple or completely absent query optimizers, which makes an approach based on independent modeling (v. probing the system) preferable because an accurate model permits potentially fuller optimization compared to what a limited number of probes can hope to achieve.

The main problem addressed by this paper is the development of such a cost model that permits optimal or near-optimal selection of indexes in a cloud-based NoSQL database service. For concreteness, the paper focuses on the *AWS DynamoDB* service, and the finer details are tied to that system. However, the general approach exemplified by this work is adaptable to other services as well. Given the importance and generality of this problem, as well as the prevalence of NoSQL systems, it may seem surprising that the work reported here has not been thoroughly addressed already by earlier work. However, there does not seem to be any reported work on such a cost model despite its need as evidenced by several publications in the practitioner community [9, 6, 3]. One reason for this disparity may be the difficulty in abstracting the salient features of the cost model from the available system documentation and other resources, which are sometimes difficult to properly interpret (e.g., Page 7).

Outline: Section 2 outlines the key features of DynamoDB tables and indexes before stating the index-selection problem in this context. Section 3 presents a detailed cost model that is validated with both the specification and by experiments on a live DynamoDB instance. Related work is addressed by Section 4 and a summary appears in Section 5.

2 Tables and Index Selection

Tables In DynamoDB, there are no operations that span multiple tables (other than drastic ones such as deleting the account that owns multiple tables). Therefore it is sufficient to approach the schema-design problem, and in particular the index-selection problem on a per-table basis; such a table is denoted by T below. Each DynamoDB table is required to have a *primary key* with semantics similar to those of primary keys in relational databases: No two items in a table are permitted to have identical primary keys. However, while the primary key of a table in a relational database may be a composite key with an arbitrary number of attributes (key-columns, or key-attributes), the primary key of a DynamoDB table is very limited: It is composed of exactly one or two attributes, the first of which is called the table's *partition key* and the second (optional) is called its *sort key*. Further, the datatypes of these keys are limited to *string*, *number*, and *binary*, although DynamoDB permits richer types (such as set of integers or list of strings) for other attributes in general. The sequel uses abbreviations *PKey* for the required *primary key* of each table or index (LSI or GSI, described below), *PaKey* for the required *partition key* (also called *hash key*) component of a PKey, and *SoKey* for the optional *sort key* (also called *range key*) component of a PKey.

Secondary Indexes DynamoDB permits two kinds of secondary indexes: *Global Secondary Indexes (GSIs)* and *Local Secondary Indexes (LSIs)*. Analogously to the restricted nature of DynamoDB tables compared with relational database tables, GSIs and LSIs are very restricted in comparison with diverse indexing schemes typically found in relational database systems. Global and local secondary indexes are required to have primary keys subject to the same restrictions as those for base tables, with the exception that there may be multiple items (index entries) for a given value of a primary key. This situation is analogous to that in relational databases, where there may be multiple tuples in a base table corresponding to a single key value in an index. The indexes are required to have a primary key composed of a required partition key and an optional sort key and both keys are limited to single attributes of scalar types. An LSI is subject to the additional restriction that its *PaKey* must be identical to the *PaKey* of its base table (and that its *SoKey* must be distinct from the base table's *SoKey*).

Every index (GSI or LSI) includes, as a minimum, the key attributes of the index along with the key attributes of the base table. Queries using an LSI are permitted to request base-table attributes that are not projected onto the index. DynamoDB automatically fetches the additional attributes from the base table

for each index item that matches a query, at additional cost. In contrast, queries on GSIs are simply not permitted to request base-table attributes that are not projected onto the index.

Index Selection The *index selection problem* is that of determining the set of indexes (combination of LSIs and GSIs) that minimizes the monetary cost of a deployment given an expected workload (composed of item reads, queries, insertions, deletions, and update operations).

3 Costs of Deployment

The cost model used by DynamoDB accounts for read and write operations separately. An orthogonal dimension is whether service is provided in *provisioned capacity mode* or *on-demand mode*. In the provisioned capacity mode, read and write operations are billed based on *Read Capacity Units (RCU)* and *Write Capacity Units (WCU)* respectively. In the on-demand mode, they are billed based on *Read Request Units (RRU)* and *Write Request Units (WRU)*, respectively.

For brevity, the following describes the cost of only *strongly consistent read* operations. For transactional reads, the costs are doubled while for *eventually consistent reads*, the costs are halved. (However, access via GSIs are only at the *eventually consistent* level.)

Reading Items From Tables Reading items by specifying the primary key (*PaKey* and, if defined for the table, the *SoKey* as well) is accomplished by two operations that are essentially equivalent from the monetary cost perspective: `GetItem` and `BatchGetItem`. In order to simplify the presentation by reducing the number of cases that need separate descriptions, we adopt the convention that if a table does not define a *SoKey* then a requirement to provide it is vacuously met. With this convention, we may consider all read-item operations as providing both *PaKey* and *SoKey* of a table.

A `GetItem` operation reads a single item (or none, if there is no item with the requested primary key) at a cost of 1 RCU times the size of the item rounded up to the nearest nonzero multiple of a blocking factor b_r that is currently set at 4 KB (4000 bytes). (Individual items may have varying sizes up to a maximum of 400 KB per item.) The cost of a `GetItem` that reads an item (or probes for its presence even if it is absent) in table T with *PaKey* value p and *SoKey* value s is simple: $c_g(T, p, s) = \max\{1, \lceil b(T(p, s))/b_r \rceil\}$ using the notation $T(p, s)$ to denote the unique item, if any, in table T with *PaKey* value p and *SoKey* value s. The function $b(X)$ denotes the size in bytes of the set X of items in general. (Here X is either the empty set or a singleton.) Due to the nonrelational nature of DynamoDB tables, size of items in a table may vary significantly from item to item.

The closely related `BatchGetItem` operation permits up to 100 primary key values to be specified at once in order to retrieve up to 100 items with those keys from the table. Cost-wise, a `BatchGetItem` operation with l keys is completely

equivalent to l individual `GetItem` operations. The reasons for using it over the equivalent individual items include programming convenience and lower system overheads such as round-trip times. In particular, this equivalence means that the sizes of items in the batch are individually rounded up to multiples of b_r and the sum of these rounded sizes determines the size of the batch for cost purposes. That is, given a set of primary keys $K = \{(p_0, s_0), (p_1, s_1) \ldots, (p_l, s_l)\}$, The cost of a `BatchGetItem` operation with that key-set is: $c_b(K) = \sum_{(p,s)\in K} c_g(T, p, s)$ It follows that $c_b(K) \geq |K|$ (even if there are no items matching any of the keys provided).

Querying and Scanning Tables A `Query` operation allows reading of zero or more items that have identical *PaKey* values and that satisfy an optionally provided predicate on the *SoKey* (if defined by the table). If items with distinct partition keys are required, a separate query must be issued for each desired partition-key value. In contrast to `BatchGetItem`, here the associated reads are treated as a single operation for cost purposes. In particular, the sizes of the matching items (before any optional *filtering*) are first summed and then rounded up to the next multiple of b_r. As a result, queries potentially provide substantial cost savings over equivalent sequences of `GetItem` or `BatchGetItem` operations that read the same items. The cost of a query on table T with *PaKey* value p and *SoKey* predicate θ may be expressed as:

$$c_q(T, p, \theta) = \max \left\{ 1, \left\lceil \frac{\sum_{t\in T(p,\theta)} b(t)}{b_r} \right\rceil \right\}$$

A `Scan` operation may be thought of as a degenerate case of a `query` that reads all items in a table. For cost computation purposes, a `Scan` incurs the cost for all items in the table whereas a typical, selective `Query` would incur the cost of reading only a small fraction of those. A scan is also notable for being the only operation that permits retrieval of items without specifying a *PaKey*. The cost of a scan is expressed simply as:

$$c_s(T) = \max \left\{ 1, \left\lceil \frac{\sum_{t\in T} b(t)}{b_r} \right\rceil \right\}$$

Other aspects of the above operations, such as *filtering* (server-side post-processing) the items matching a query to limit those that are returned to the invoking application, or returning only the count of items matching a query, do not reduce the costs. For instance, determining the number of items with a given partition key value (a *count* operation in conventional database systems) incurs the same cost as retrieving all the items with that partition key value. Requests to read non-existing items (e.g., a `GetItem` that specifies a primary key that does not match any items in the table) incur the same cost as if the item were present and read.

Consider a query on base table T using LSI (local secondary index) I with *PaKey* value p, *SoKey* predicate θ, and attribute set A. Let $attr(I)$ denote the

set of *projected attributes* of index I. The set of items in table T that have *PaKey* p and a *SoKey* value s satisfying predicate θ is denoted by $T(p, \theta)$. This notation extends the earlier notation of $T(p, s)$ used to denote $T(p, \theta_s)$ where θ_s is the predicate that tests for equality with s. The similar set of items for an index I is denoted by $I(p', \theta')$, where the primed variants of the arguments are used as a reminder that the key attributes of indexes differ from those of their base tables. The *base-table PaKey* value and *base-table SoKey* value of an index item i are denoted by i_p and i_s, respectively. If i is an item in an LSI then $i_p = p'$ because they are values of the same attribute (due to the restriction on LSIs), but $i_s \neq s'$ in general. This notation allows generalizing to GSIs later: If i is an item in a GSI, both i_p and i_s are, in general, different from the values p' and s' of that item's *index PaKey* and *index SoKey*, respectively. The notation $[P]$, where P is a predicate, uses *Iverson brackets*: It evaluates to 1 if the P is true and to 0 otherwise. With these conventions, the cost of a query using LSI I is expressed as follows:

$$c_q(T, I, p', \theta', A) = \max\left\{1, \left\lceil \frac{\sum_{i \in I(p', \theta')} b(I, i)}{b_r} \right\rceil \right\} + [A \not\subseteq \mathit{attr}(I)] \cdot \sum_{i \in I(p', \theta')} \max\left\{1, \left\lceil \frac{b(T, t(i_p, i_s)}{b_r} \right\rceil \right\}$$

The summation in the second term, over items matching the query, is notable for being outside (after) the item-wise size computation. For queries returning a large number of small items with at least one attribute not present in the used index, this term is likely the dominant contributor to cost.

Despite the DynamoDB restriction on GSI queries that disallows access to base-table attributes that are not projected onto the index, the above equation may be used to model the cost of index queries that use either an LSI or a GSI with the understanding that in the GSI case, the additional accesses modeled by the second term are made separately by the driving application and not automatically by DynamoDB as part of the same query operation.

Write Costs Write operations use a blocking parameter b_w that is analogous to the parameter b_r for read operations. Currently $b_w = 1$ KB (1000 bytes), in contrast to $b_r = 4$ KB. The cost of *deleting or inserting* an item with *PaKey* p and *SoKey* s from or into table T that is due to the table itself is simply one unit, or the next higher multiple of b_w for larger items. The max is present to account for the minimum cost that is incurred even on requests to delete non-existing items. The subscript t on c_{dt} indicates that this is only the cost component attributed to the base table.

$c_{dt}(T, p, s) = c_{it}(T, p, s) = \max\{1, \lceil b(T(p, s))/b_w \rceil\}$

More interesting is the additional cost incurred by the presence of indexes on T. That cost depends on whether its values for the index-key attributes are null and may be expressed as follows for a GSI $I_{P'S'}$, as suggested by the g suffix on c_{dg}, using the Iverson bracket notation. $c_{dg}(T_{PS}, I_{P'S'}, t) = c_{ig}(T_{PS}, I_{P'S'}, t) = [t_{P'} \neq \perp \wedge t'_S \neq \perp] \max\{1, \lceil b(T(p, s))/b_w \rceil\}$

Updating an item from t to u may incur a cost of 0, 1, or 2 units times the size-based scaling, depending on whether the index-key attributes have non-null values and, additionally, whether they are modified by the update operation. It is convenient to express it using the costs c_{ig} and c_{dg} defined above.
$c_{ug}(T_{PS}, I_{P'S'}, t, u) = c_{ig}(T_{PS}, I_{P'S'}, u) + [t_{P'} \neq u_{P'} \vee t_{S'} \neq u_{S'}] \cdot c_{dg}(T_{PS}, I_{P'S'}, t)$

The description of LSI write costs (capacity units) on of the DynamoDB developer guide [1, page 490], when compared with the analogous description for GSI write costs on [1, page 454], may be read as implying that an LSI incurs no cost when an item is updated to change only a non-key projected attribute. However, experimentation reveals that in fact the cost incurred in such a case is identical to that incurred by a similar GSI. Then the write costs of an LSI may be expressed using the above equation used for GSIs, with a couple of clarifications: First, since $P = P'$ for LSIs, and a base-table *PaKey* cannot be omitted, the predicate $P' \neq \perp$ always evaluates to true. Second, the cost in the provisioned capacity mode is allocated to the base table as is the case for all LSI costs.

Storage Costs Let $attr(I_{P'S'})$ denote the attributes (names) of index I (with index *PaKey* P' and index *SoKey* S'), including all key and nonkey attributes. If the base table for $I_{P'S'}$ is T_{PS} (with *PaKey* P and *SoKey* S) then $\{P, S, P', S'\} \subseteq attr(I_{PS}) \subseteq attr(T_{PS})$. Let b_o denote a constant overhead, in bytes, assessed per index item; its value at the time of writing is 100 [1]. The storage used by the GSI $I_{P'S'}$ on base table T_{PS} may then be expressed as follows:

$$b(T_{PS}, I_{P'S'}) = \sum_{\substack{t \in T_{PS} \\ t_{P'} \neq \perp \\ t_{S'} \neq \perp}} \sum_{A \in attr(I_{P'S'})} b(t_A) + b_o$$

Recall that t_A (the value of the attribute named A in item t) may be absent (modeled as null, $\perp$) except when A is a key attribute (of the base table, not necessarily index). Items that have a null for an index's *PaKey* or, if used by the index, its *SoKey*, do not contribute to index size. At one extreme, if none of the items in T have non-null values for these attributes then the storage cost of the index is 0. At the other extreme, if every item in the base table has non-null values for these attributes, and if the index projects all attributes from the base table, then the additional storage used by the index is $b(T) + b_o \cdot |T|$, where $b(T)$ is the size in bytes of the base table and $|T|$ is its cardinality (number of items).

We may use the above expression of GSI storage costs for LSIs as well, with the understanding that $P = P'$ when it is used for LSIs. However, LSIs (and their base tables) are also subject to a limit on the sizes of *item collections* from which GSIs (as well as their base tables, if they have no LSIs) are exempt. Briefly, an item collection refers to all items in a base table and its LSIs that have the same *PaKey* value. The aggregate size (in bytes) of items in an item collection can be no greater than 10 GB. This limit on item collection sizes adds another design constraint for LSIs, in addition to the limit of at most five LSIs per base table. It is possible, for instance, that an otherwise advantageous choice

of an LSI must be abandoned due to the item-collection constraint. DynamoDB design guidelines suggest ways to reduce sizes of item collections. However, since a single LSI may (depending on projected attributes) approximately double the size of an item collection, the constraint may be limiting when certain LSIs are considered even if the base table's item collections are well below the limit.

4 Related Work

The difficulty of reliably predicting monetary costs of DynamoDB deployments has been widely reported in the practitioner literature [6, 9, 10] including billing surprises of large magnitude.

The authoritative DynamoDB documentation [1] provides detailed, albeit sometimes difficult to interpret information on how operations map to monetary costs. It also provides heuristic guidance on recommended practices for index selection but no systematic algorithms or numerical formulas assist with the process. Index selection is a component of a larger scheme for query optimization in DynamoDB and related systems [5].

The more conventional index selection problem in relational DBMS has received significant attention [4]. However, although there are some similarities, such as the combinatorial explosion in the number of potential indexing schemes with increase in the number of attributes of base tables, the crux of the problem is very different in the conventional relational and cloud NoSQL environments. In the former, a major difficulty is determining the effect of indexes on the query plans used by the query optimizer and their costs. Given the level of sophistication of query optimizers in competitive relational database systems, the preferred approach has been to probe the optimizer for that information instead of trying to model the actions of the optimizer. However, in a NoSQL cloud database setting, the optimizers are either very simple or completely absent, so that modeling is a viable option. On the other hand, the cost model itself is more complicated and irregular.

The problem of materialized view selection [8] may be viewed as an index selection problem generalized to more expressive indexes (specified using SQL queries or similar). In the other direction, physical database design [7] goes beyond indexes and into other systems aspects affecting database performance.

DQL [2] is a notable effort in regularizing the query-language interface of DynamoDB to a syntax that is close to SQL, albeit with significant restrictions.

5 Conclusion

Selecting an appropriate set of secondary indexes for a database given an expected workload of read operations (individual item reads and bulk queries) and write operations (insertions, deletions, updates) has a major impact on the performance of database systems and applications in diverse environments. For cloud-based NoSQL database services, as exemplified by AWS DynamoDB, different selections of secondary indexes can incur monetary costs that differ by

orders of magnitude, making the index selection problem in such environments especially significant. Despite the high practical significance and monetary implications of this problem as evidenced by a multitude of reports in the practitioner literature, the problem has received very little attention in the research community. This paper presents the first comprehensive formalized cost model for secondary index selection in DynamoDB. An important feature of this cost model is that it has been carefully verified with not only the system specification documents but also with experiments on a live system (by comparing actual reported costs with those predicted by the model). This verification allows the model to be used with confidence for further research as well as in practice, where the costs map directly to usage charges.

Ongoing work is exploring simplifications of the cost model in order to allow easier application at the potential cost of less accurate results and higher than optimal monetary cots. It is also exploring a simlar approach applied to other NoSQL database services.

Acknowledgments This work was partly supported by the US National Science Foundation and benefited from the reviewers' comments.

References

1. Amazon Web Services: Amazon DynamoDB: Developer guide. `https://docs.aws.amazon.com/dynamodb/` (2019), API Version 2012-08-10
2. Arcangeli, S.: DQL—DynamoDB query language: A simple, SQL-ish language for DynamoDB. System documentation (Mar17 2018), `https://dql.readthedocs.io/en/latest/index.html`
3. Brazeal, F.: Why Amazon DynamoDB isn't for everyone. A Cloud Guru blog (Nov18 2017), `https://read.acloud.guru/why-amazon-dynamodb-isnt-for-everyone-and-how-to-decide-when-it-s-for-you-aefc52ea9476`
4. Chaudhuri, S., Narasayya, V.R.: Self-tuning database systems: A decade of progress. In: Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB). pp. 3–14. ACM (Sep23-27 2007)
5. Chawathe, S.S.: Cost-based query-rewriting for DynamoDB (work in progress). In: Proceedings of the 17th IEEE International Symposium on Network Computing and Applications (IEEE NCA 2019). Cambridge, MA (Sep26-28 2019), to appear
6. Guerrero, J.: How DynamoDB's pricing works, gets expensive quickly and the best alternatives. YugabyteDB Blog. `https://blog.yugabyte.com/dynamodb-pricing-calculator-expensive-vs-alternatives/` (Jul18 2018)
7. Lightstone, S.S., Teorey, T.J., Nadeau, T.: Physical Database Design : The Database Professional's Guide to Exploiting Indexes, Views, Storage, and More. Elsevier Science & Technology, San Francisco, California (2007)
8. Mami, I., Bellahsene, Z.: A survey of view selection methods. SIGMOD Rec. **41**(1), 20–29 (Apr 2012). https://doi.org/10.1145/2206869.2206874
9. Ranganathan, K.: 11 things you wish you knew before starting with DynamoDB. yugabyteDB blog (Jul10 2018), `https://blog.yugabyte.com/11-things-you-wish-you-knew-before-starting-with-dynamodb`
10. Roussel, A., Branson, R.: The million dollar engineering problem. segment blog (May14 2017), `https://segment.com/blog/`