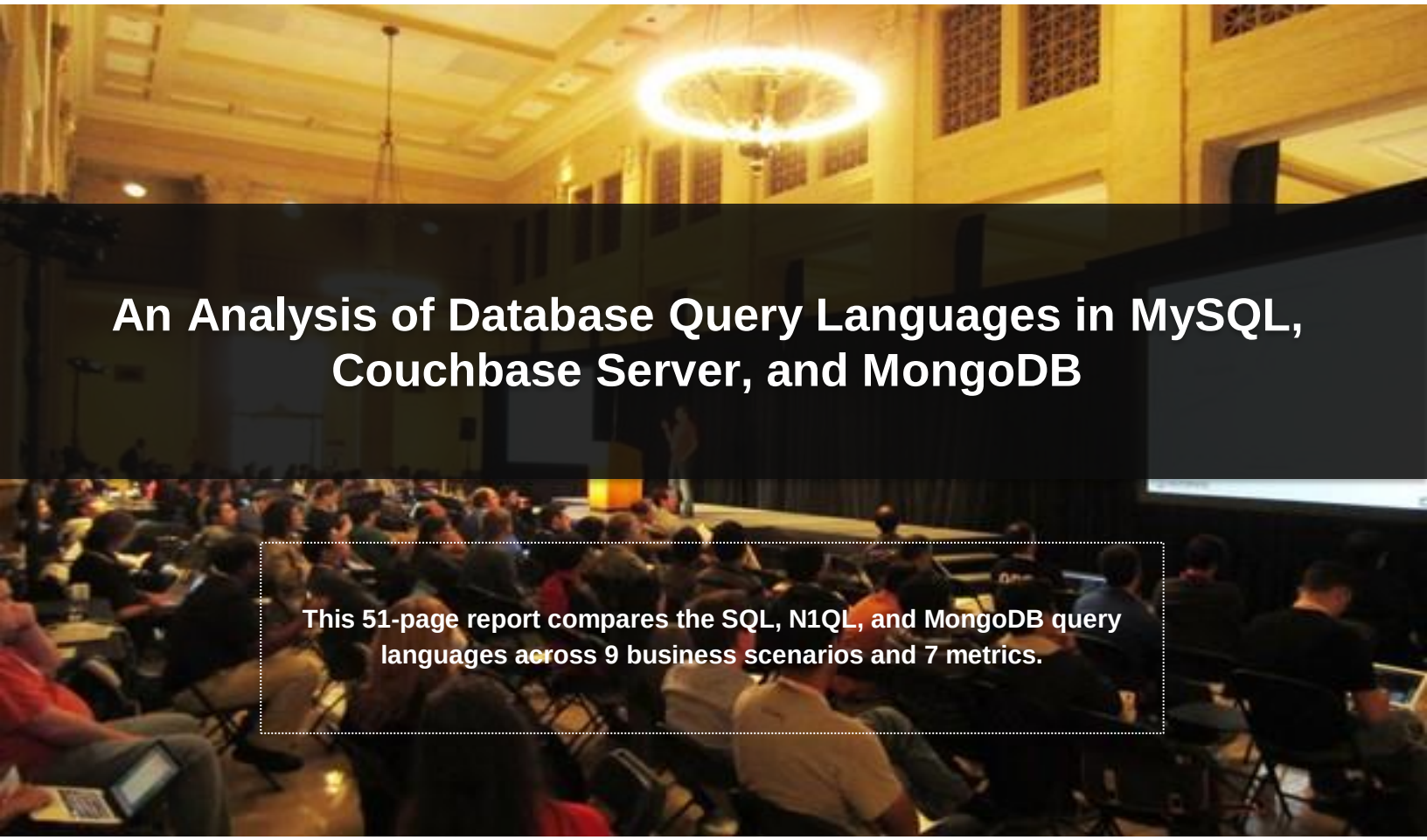
ALTOROS™



An Analysis of Database Query Languages in MySQL, Couchbase Server, and MongoDB

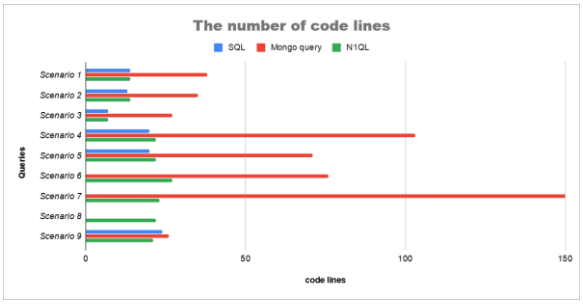
This 51-page report compares the SQL, N1QL, and MongoDB query languages across 9 business scenarios and 7 metrics.

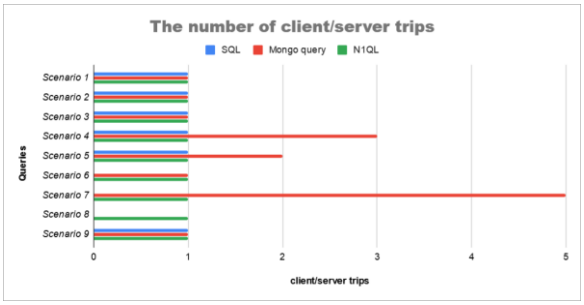
By **Artsiom Yudovin**, Data Engineer
Uladzislau Kaminski, Senior Software Engineer

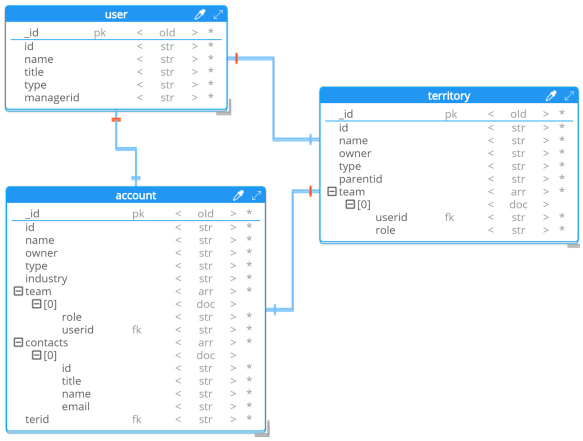
Q3 2019

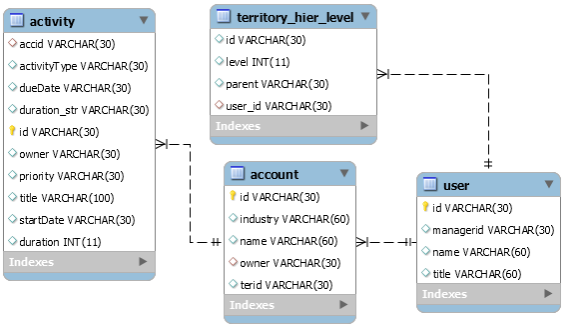
Table of Contents

Featured figures:









1. Introduction

All of today's online activities—from browsing the news, interacting on social media, looking for a product, booking a flight, or making a purchase—have been made possible by the recent innovations in database technology. The applications that support these activities are now operating at speeds and throughput levels rarely seen before. Organizations no longer rely on a single database vendor to meet all their needs. Instead, databases are specialized for different uses, such as streaming, analytics, or structured and unstructured data processing. These databases can be in-memory, deployed to the cloud, and replicated across multiple geographical regions for high availability, improved latency, or specific security needs. The most recent DBTA report has over 28 database categories that customers can choose from to get the best fit.

All the databases out there share a common objective, which is to provide the most efficient data manipulation mechanism, so applications can efficiently query the data being managed. A traditional relationship database management systems (RDBMSes), such as Oracle, SQL Server, or DB2, use SQL as a standard to access data. However, most of the NoSQL databases rely on a proprietary language or APIs.

This report will highlight how SQL remains the language being emulated by leading NoSQL databases, such as MongoDB and Couchbase. MongoDB provides a MongoDB query, an API approach that allows users to filter, join, aggregate, order, and project the query result. Couchbase uses the N1QL language to extend ANSI SQL, a standard for the SQL language, to achieve the same goals.

This report provides a comparative analysis of a MongoDB query, the N1QL language, and the SQL language using MySQL as a reference for it. Each query language is then tested against nine business scenarios, ranging from simple queries to complex aggregations that use a simple enterprise activity management data set. For each scenario, the three languages are rated against the same set of criteria to compare their relative power and simplicity.

A note on methodology: for criteria based on measurable data, scores were applied based on a real-world experience using the products under evaluation, as well as on regularly conducted benchmarks. For criteria based on qualitative data (e.g., installation and maintenance procedure), scores were applied based on an in-depth review of the documentation, feedback from solution vendors and engineering teams at enterprises, and our own development and production experience.

All the examples of queries and database dumps, which can help to deploy and run all the scenarios from this report can be found in this GitHub repository.

2. Key findings

There were several points of interest uncovered during the comparative analysis. These include fundamental distinctions between the three database languages in how their queries are structured, as well as differences in query syntax and efficiency.

The first aspect that stood out in this analysis is the difference between the declarative approach of MySQL and N1QL compared to the procedural approach of a MongoDB query.

A declarative versus procedural approach

The declarative approaches of MySQL and N1QL allow users to express a desired outcome, while a database server will decide how to obtain that result. A MongoDB query allows a user to write a pipeline code for how the result should be achieved, which is implicitly specified by the different stages in the aggregation pipeline. Scenarios 3.1–3.3 (described in the sections below) show how a MongoDB query can be quite effective with the procedural approach, enabling a user to explicitly set the order of an operation–particularly for filtering with `$match` and then a collection join with `$lookup`. Here, the order of operations can significantly alter a query's response time.

In the case of MySQL and N1QL, a user does not have to specify the order. This declarative approach relies on a database query optimizer to understand that it is more efficient to perform a filtering operation before a join operation. This feature is well supported by the Couchbase query optimization.

The second salient observation in the report is the syntax of MySQL and N1QL compared to that of a MongoDB query.

A SQL standard versus a proprietary API

When it comes to comparing the readability or skill level between the two implementations, opinions can be somewhat subjective. There are those who prefer the procedural nature of a MongoDB query. Becoming proficient in the MongoDB API is no more challenging than it is for MySQL as both take time to master. However, many developers are already proficient in MySQL, because it has been around for more than four decades. This familiarity could be a factor with organizations that want to leverage their existing skills or want an easier time finding new developers with the skill sets needed.

The third point of emphasis in this comparative analysis is how the queries are processed in each of the implementations.

A number of application-to-server round trips

With both MySQL and N1QL, the query is processed entirely in a database server. In effect, only a single submission of the MySQL or N1QL query to the server is needed for the result to come back. With MongoDB, more complex queries (Scenarios 3.4, 3.5, and 3.7, which are described in the sections below) would require as many as five trips between the client application and the MongoDB server. This can significantly slow down the overall response time and also requires more resources in the client applications. The need to break the query down into multiple components before passing the result into the next call would also add more complexities to the client applications.

There is also a final subtle distinction that the report was not able to show due to the complexity that would be required in the MongoDB query's code. This distinction is in the ability to join data sets in a sharded collection.

Join support in a sharded collection

Couchbase's N1QL supports all the different types of the ANSI joins. The `JOIN` operation can happen on any sharded Couchbase bucket. A MongoDB query's `$lookup` only works with unsharded collections. There are two ways to overcome this limitation in a MongoDB query: code the `$lookup` logic in the client application or avoid the `$lookup` altogether by designing a more denormalized data model. This limitation could be a major hurdle for organizations that want to migrate their existing RDBMS-based applications to NoSQL. A `JOIN` operation using Couchbase's N1QL is agnostic to how the underlying data is organized and managed.

Figure 2.1 depicts the number of code lines in the query for each scenario included in this report. This parameter influences most of the chosen criteria, because all the limitations and disadvantages of query languages will increase the lines of code.

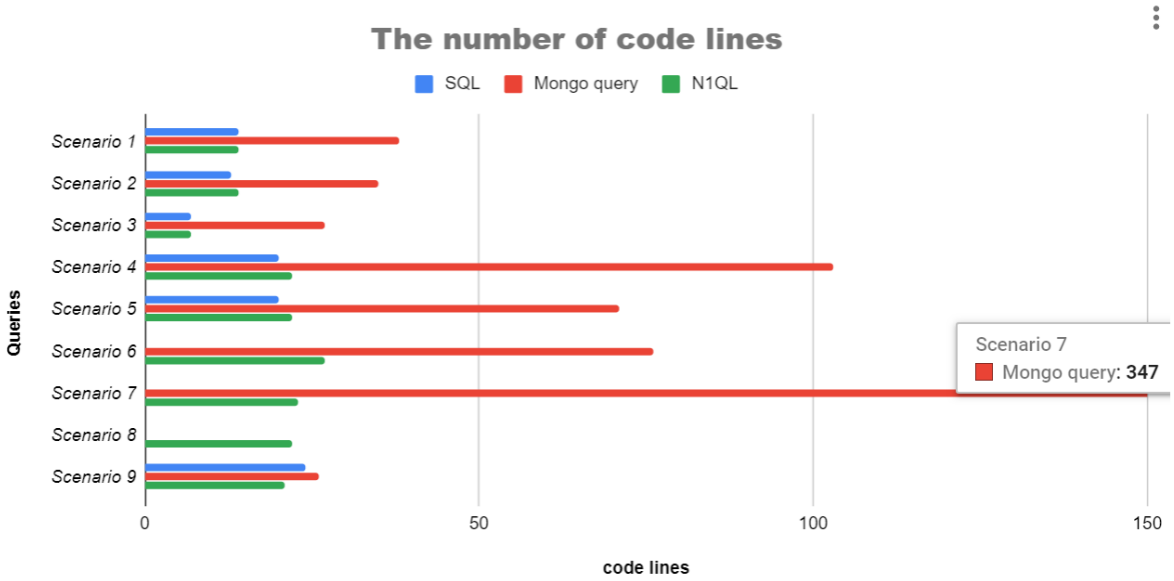


Figure 2.1 Total code lines in the query for each of the nine scenarios

The number of trips between the client application and the database server are shown in Figure 2.2. This parameter will impact the performance of the queries, because it can lead to potentially large amounts of data being transferred between a client and a server.

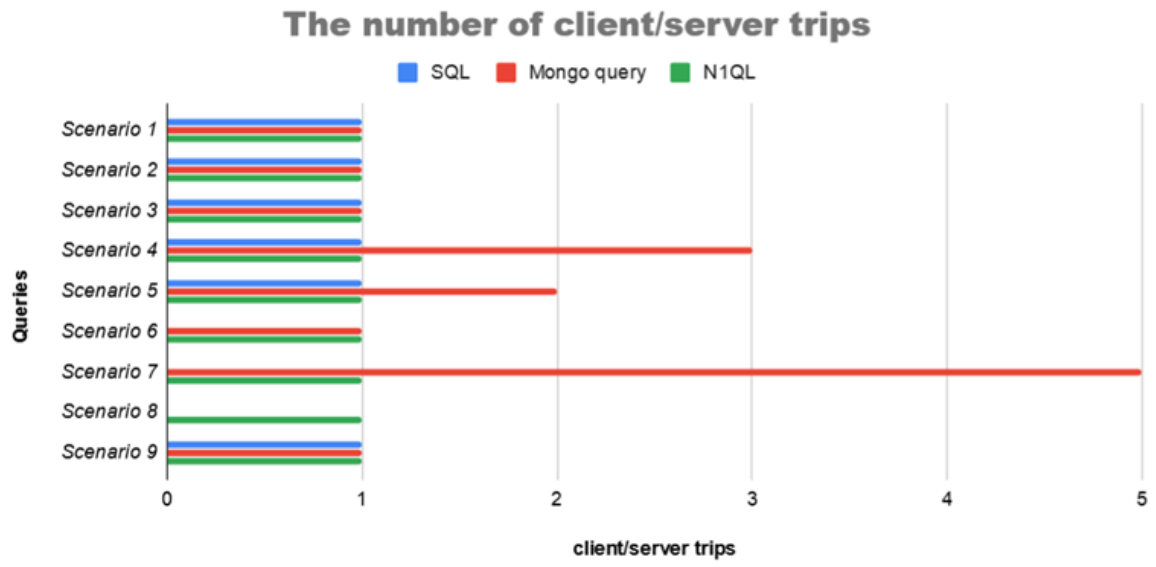


Figure 2.2 Total application-to-server round trips in each of the nine scenarios

Table 2.1 demonstrates the average marks for each database. The average mark was calculated from the criteria result for each scenario.

Table 2.1 Average scores for each metric

Criteria	MySQL	N1QL	MongoDB query
Simplicity	8.3	8.5	6.4
Readability	9	9.1	6.8
Expressiveness	9	8.8	6.3
Flexibility	8.8	9	7.4
Skills availability	7.7	6.7	5.6
Average score	**8.6**	**8.4**	**6.5**

Table 2.2 shows two physical criteria. The (`l / t`) value in each database implementation represents:

- `l` is the total number of code lines in the query. This parameter influences most of the chosen criteria, because all the limitations and disadvantages of query languages will increase the number of code lines.
- `t` is the number of trips between a client app and a database server. This parameter will impact the performance of the queries, because it can lead to potentially large amounts of data being transferred between a client and a server.

Table 2.2 Lines of code and total application-to-server round trips in each scenario

Scenario	Query description	MySQL	N1QL	MongoDB query
3.1. Meeting customers	To prepare for customer meetings that I will be attending next week, I want to get a list of all the customers to attend the meetings and their contacts.	14 / 1	14 / 1	38 / 1
3.2. Regional sales management	I am a Regional Sales Manager for the *C-Suite Sellers* territory. I want to get all accounts assigned to this territory and the account team members.	13 / 1	14 / 1	35 / 1
3.3. Sales activities	Determine the top 10 industries from our customers based on the 2018 sales activities.	7 / 1	7 / 1	27 / 1
3.4. Sales organizations	I run a sales organization for the *Aggressive Achievers* territory. I want to find out how much time we spent talking to the accounts assigned to this territory for *Q3FY19*.	20 / 1	22 / 1	103 / 3
3.5. A sales task report	This scenario shows how the number of sales-related tasks have changed a month over a month during the year 2018.	20 / 1	22 / 1	71 / 2
3.6. A skill set report	The company is performing an analysis on the sales team skill sets/roles in the current sales	N/A	27 / 1	76 / 1

	organization. It needs to identify all the territories where there is only a single person with a specific role/skill set, and that the territory handles more than five accounts.			
3.7. Search contacts	A query to review all the presentations that we have conducted with the customers in *CY19Q4*. The query needs to show the time we spent for each meeting, the running count, and the percentage of time for the meeting over the total time we spent talking to the customer. Furthermore, the query needs to calculate `high_touch_rank`, which is the percentage of the customer's contacts who attended the meeting against the total number of the customer's contacts.	N/A	23 / 1	347 / 5
3.8. Calling Google Natural Language API	In order to find a based on the most positive reviews, you could read through all the reviews for all the hotels or leverage Google Natural Language API to analyze the sentiment of the reviews. The query should return top 10 hotel reviews based on the sentiment score.	N/A	22 / 1	N/A
3.9. Search criteria	Our goal is to identify the customer accounts and their related contacts, where a particular topic has been discussed. The search criteria may include the following information partially or in full: a meeting title, a meeting date range, customer contact details, sales team member details (participants), and a customer name.	24 / 1	21 / 1	26 / 1

3. Scenarios

In this section, each of the nine business scenarios are illustrated through relational and JSON models. Additionally, query implementations for each database language are provided along with concluding remarks and summary for metrics.

3.1. Meeting customers

To prepare for customer meetings that I will be attending next week, I want to get a list of all the customers to attend the meetings and their contacts.

Figure 3.1.1 demonstrates a relational model for the current scenario.

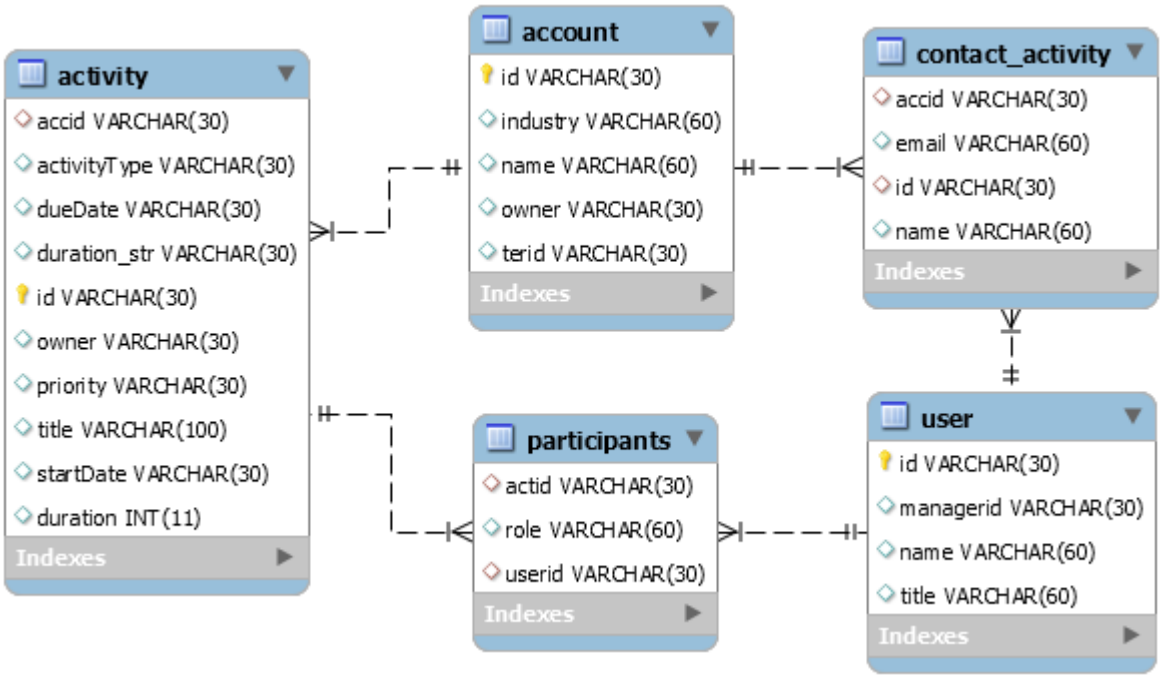


Figure 3.1.1 A relational model for the Meeting customers scenario

Figure 3.1.2 demonstrates a JSON model for the current scenario.

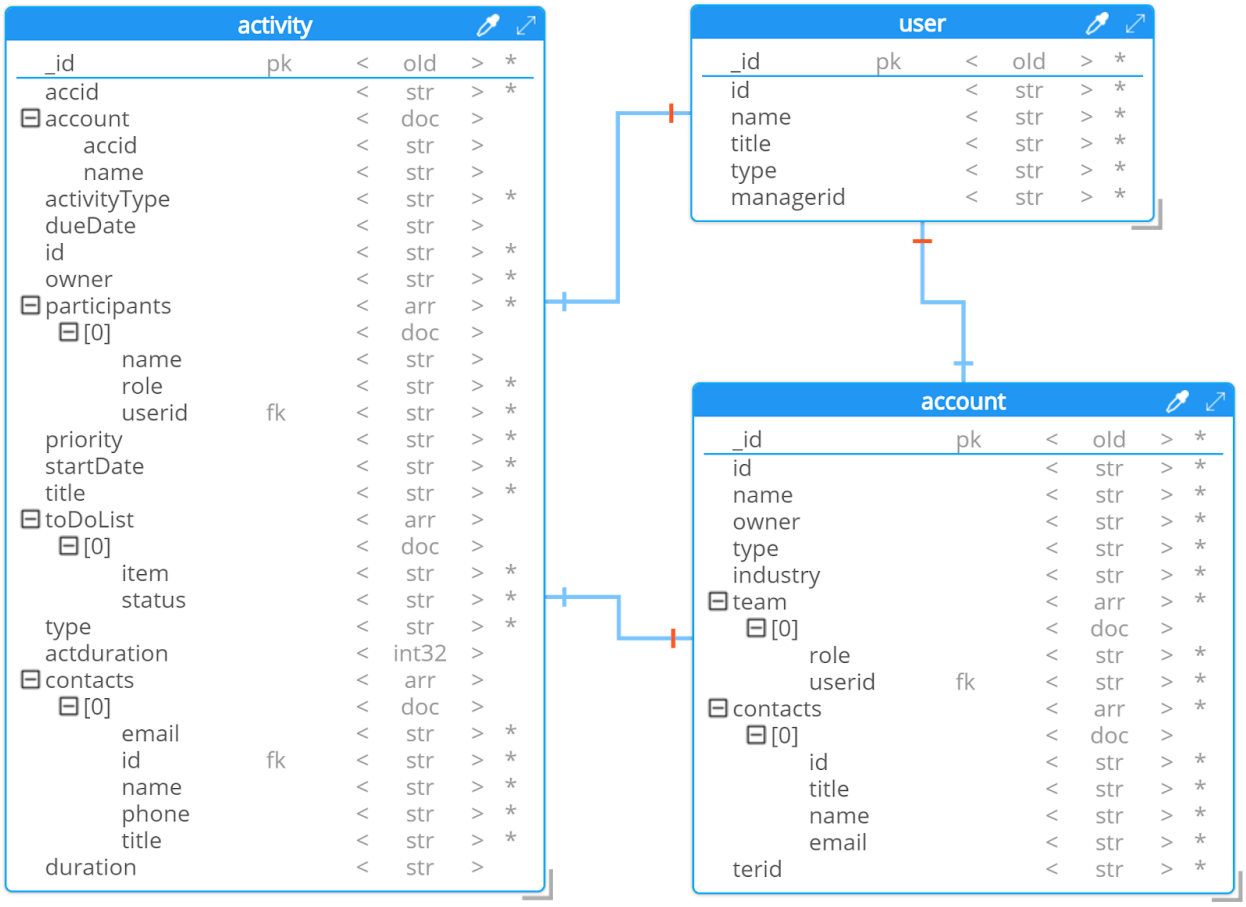


Figure 3.1.2 A JSON model for the Meeting customers scenario

The current scenario can be achieved with all the three languages.

The query described in Listing 3.1.1 is an implementation of the current scenario for MySQL.

Listing 3.1.1 An SQL implementation for the Meeting customers scenario

```
SELECT a.id,
 a.title meeting,
 c.name customer,
 a.startdate,
 cn.name contact_name,
 cn.title contact_title,
 cn.email contact_email
FROM  activity a
 INNER JOIN account c ON (a.accid = c.id)
 INNER JOIN contact_account cn ON (c.id = cn.accid)
 INNER JOIN participants p ON (a.id = p.actid)
WHERE a.activitytype = 'Appointment'
   AND a.startdate BETWEEN '2019-01-01'  AND '2019-01-31'
   AND p.userid='user73'
```

The query described in Listing 3.1.2 is an implementation of the current scenario for N1QL.

Listing 3.1.2 An N1QL implementation for the Meeting customers scenario

```
SELECT a.id,
 a.title meeting,
 c.name customer,
 a.startdate,
 cn.name contact_name,
 cn.title contact_title,
 cn.email contact_email
FROM  crm a
 INNER JOIN crm c ON (a.accid = c.id  AND c.type = 'account' )
 UNNEST c.contacts AS cn
WHERE  a.type = 'activity'
 AND a.activitytype = 'Appointment'
 AND a.startdate BETWEEN '2019-01-01' AND '2019-01-31'
 AND ANY p IN a.participants SATISFIES p.userid = 'usr73' END
```

The query described in Listing 3.1.3 is an implementation of the current scenario for a MongoDB query.

Listing 3.1.3 A MongoDB query implementation for the Meeting customers scenario

```
db.activity.aggregate([
  {
     $match: {
       $and: [
         { "participants.userid": { $eq: 'usr73' } },
         { activityType: { $eq: 'Appointment' } },
         {
          startDate: {
             $gt: '2019-01-01',
             $lt: '2019-01-31'
          }
        }
        ]
       }
    },
    {
        $lookup:
          {
            from: "account",
            localField: "accid",
            foreignField: "id",
            as: "account_docs"Mongo
          }
    },
    { $match: { "account_docs": { $ne: [] } } },
    { $unwind: "$account_docs" },
    { $unwind: "$account_docs.contacts" },
    {
       $project: {
          _id: 0, title: 1,
```

```
            startDate: 1,
            accountname: "$account.name",
            contactname: "$account_docs.contacts.name",
            contacttitle: "$account_docs.contacts.title",
            email: "$account_docs.contacts.email"
        }
    }
]).pretty();
```

Summary

Using the queries above, all three databases get the same results. This scenario is simple, and the business value of this query is obvious. Each database provides all the necessary functionality for implementation.

N1QL and a MongoDB query also support the join operation as MySQL. N1QL provides the `JOIN` operation, while a MongoDB query has `LOOKUP`. Based on the nature of the databases, N1QL and a MongoDB query use less `JOIN` operations than MySQL, because N1QL and a MongoDB query are document-oriented databases. Due to this nature, N1QL and a MongoDB query support extra functionality, which can help to decrease the number of `JOIN` operations. N1QL has the `UNNEST` operation, and a MongoDB query offers the `UNWIND` operation. The `UNNEST` and `UNWIND` operations allow performing a join of the nested array with its parent object. This makes sense for N1QL and a MongoDB query, because there are embedded documents in our JSON model.

This query is not difficult to achieve for each database. N1QL and a MongoDB query have fewer JOIN operations, because they are different in nature to MySQL. A MongoDB query is also less expressive.

Table 3.1.1 Metrics for the Meeting customers scenario

Criteria	MySQL	N1QL	MongoDB query
Simplicity	9	9	9
Readability	10	10	10
Expressiveness	9	9	8
Flexibility	9	9	9
Skills availability	9	7	7
A number of code lines	14	14	38
A number of client/server trips	1	1	1

3.2. Regional sales management

I am a Regional Sales Manager for the *C-Suite Sellers* territory. I want to get all the accounts assigned to this territory and the account team members.

Figure 3.2.1 demonstrates a relational model for the current scenario.

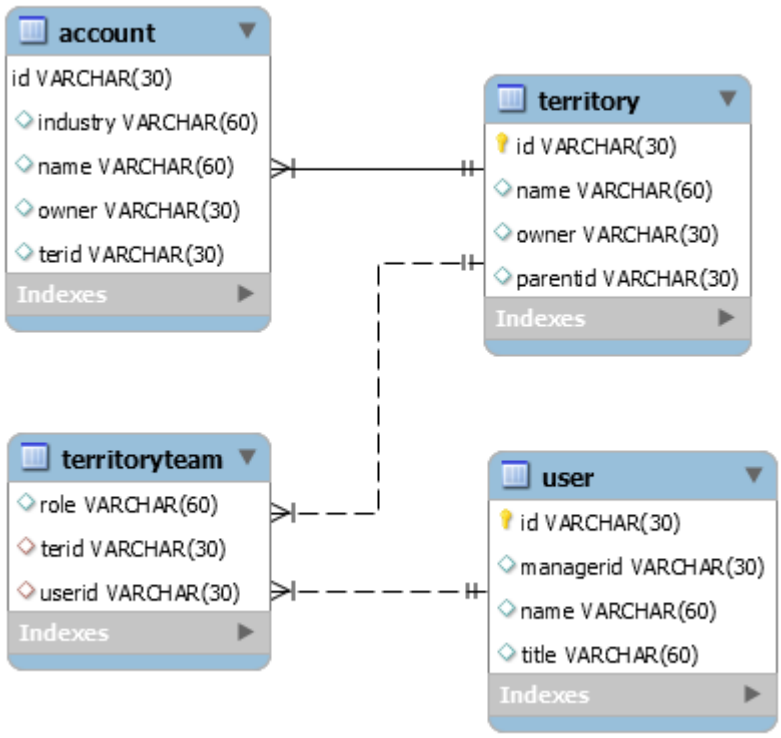


Figure 3.2.1 A relational model for the Regional sales management scenario

Figure 3.2.2 demonstrates a JSON model for the current scenario.

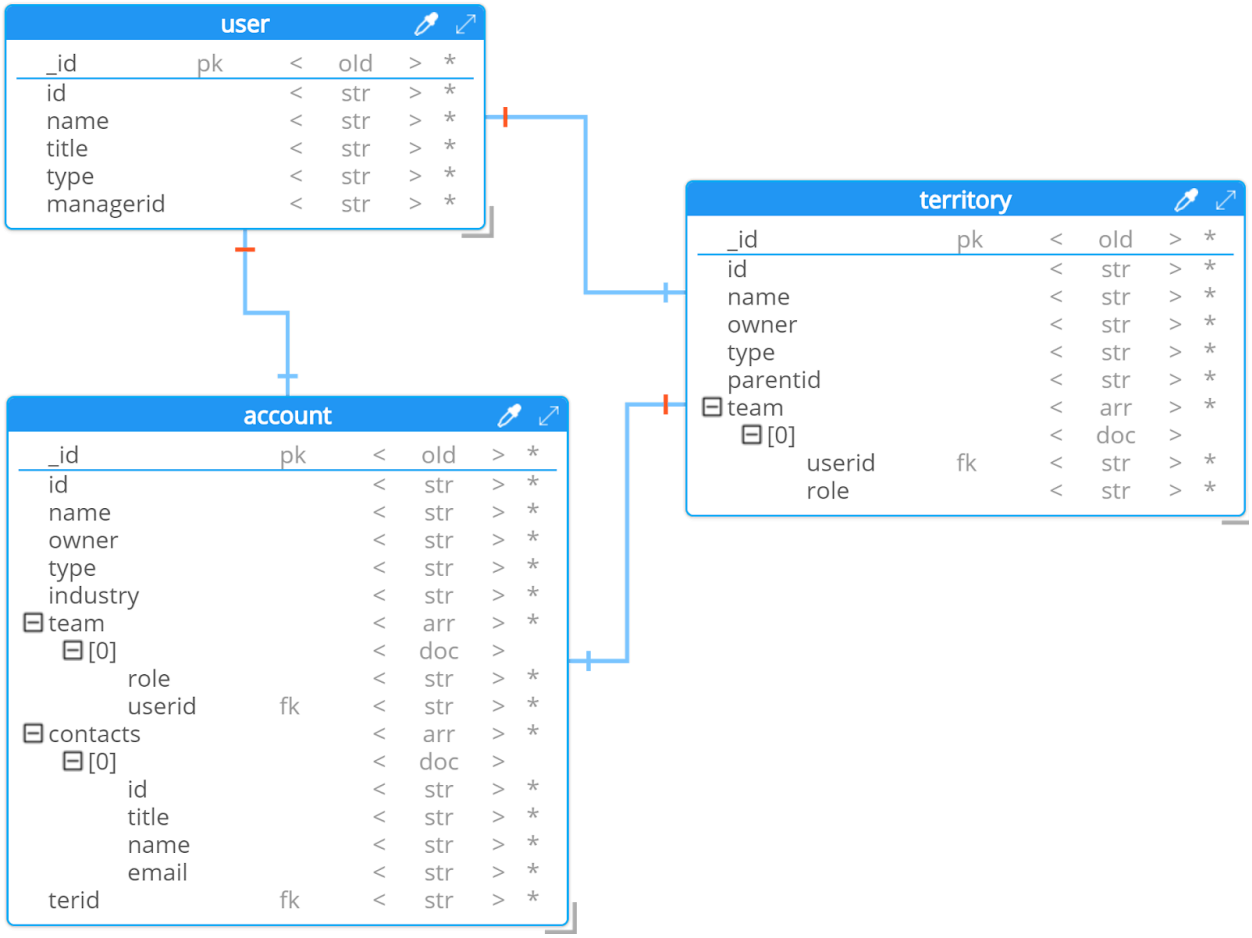


Figure 3.2.2 A JSON model for the Regional sales management scenario

The current scenario can be achieved with all three languages.

The query described in Listing 3.2.1 is an implementation of the current scenario for MySQL.

Listing 3.2.1 An SQL implementation for the Regional sales management scenario

```
SELECT ter.name `Territory`,
 cus.name Customer,
 u.name `Territory team member`,
 u.title `Member Title`
FROM  crm.territoryteam tt
 INNER JOIN crm.territory ter ON tt.terid = ter.id
 INNER JOIN crm. `user` u ON tt.userid = u.id
 INNER JOIN crm.account cus ON ter.id = cus.terid
WHERE   ter.name = 'C-Suite Sellers'
GROUP BY  ter.name,
 cus.name,
 u.name,
 u.title
```

The query described in Listing 3.2.2 is an implementation of the current scenario for N1QL.

Listing 3.2.2 An N1QL implementation for the Regional sales management scenario

```
SELECT  ter.name `territory`,
 cus.name customer,
 usr.name `territory team member`,
 usr.title `member title`
FROM  crm usr
 INNER JOIN crm cus ON ( ANY v IN cus.team SATISFIES usr.id = v.userid END
                                                  AND cus.type = 'account')
 INNER JOIN crm ter ON (ter.id = cus.terid AND ter.type = 'territory')
WHERE usr.type = 'user'
 AND ter.name = 'C-Suite Sellers'
GROUP BY  ter.name,
 cus.name,
 usr.name,
 usr.title
```

The query described in Listing 3.2.3 is an implementation of the current scenario for a MongoDB query.

Listing 3.2.3 A MongoDB query implementation for the Regional sales management scenario

```
db.territory.aggregate(
  { $match: { name: { $eq: 'C-Suite Sellers' } } },
  { $unwind: "$team" },
  {
     $lookup: {
          "from": "user",
          "localField": "team.userid",
          "foreignField": "id",
          "as": "teammembers"
     }
  },
  { $match: { "teammembers": { $ne: [] } } },
  { $unwind: "$teammembers" },
  {
     $lookup: {
          "from": "account",
          "localField": "id",
          "foreignField": "terid",
          "as": "terraccount"
     }
  },
  { $match: { "terraccount": { $ne: [] } } },
  { $unwind: "$terraccount" },
  {
     $group: {
       "_id": {
          terrname: "$name",
          accountname: "$terraccount.name",
          teammember: "$teammembers.name",
          membertitle: "$teammembers.title"
```

```
        }
      }
    },
    { $project: { _id: 1 } },
).pretty();
```

Summary

To achieve this business result, data from several tables (documents) needs to be aggregated.

MySQL provides a way to combine several tables with a `JOIN` operator. Additional conditions can be placed on the final table with the results using the `WHERE` clause. N1QL employs another approach, since it supports the `ARRAY_JOIN` operation that expands the initial document with values from other documents. Additionally, N1QL provides a way to add conditions using the `SATISFIES` operator. For a MongoDB query, several `LOOKUP` methods can be applied to collect all the necessary information from a document and the `UNWIND` operation is used to flatten the structure. The grouping operation is similar for all databases. For these queries, it makes sense to compare the complexity of creating the query and intermediate results structures. For MySQL, a large number of relations should be initiated, and it makes the building process more complex. N1QL provides a way to use fewer relations and keeps the structure easier to read. A MongoDB query does not provide the `ARRAY_JOIN` operation, and the `UNWIND` operation leads the considerable intermediate result.

Table 3.2.1 Metrics for the Regional sales management scenario

Criteria	MySQL	N1QL	MongoDB query
Simplicity	8	9	8
Readability	9	9	9
Expressiveness	9	9	8
Flexibility	9	9	9
Skills availability	9	8	8
A number of code lines	13	14	35
A number of client/server trips	1	1	1

3.3. Sales activities

Determine the top 10 industries from our customers based on the 2018 sales activities.

Figure 3.3.1 demonstrates a relational model for the current scenario.

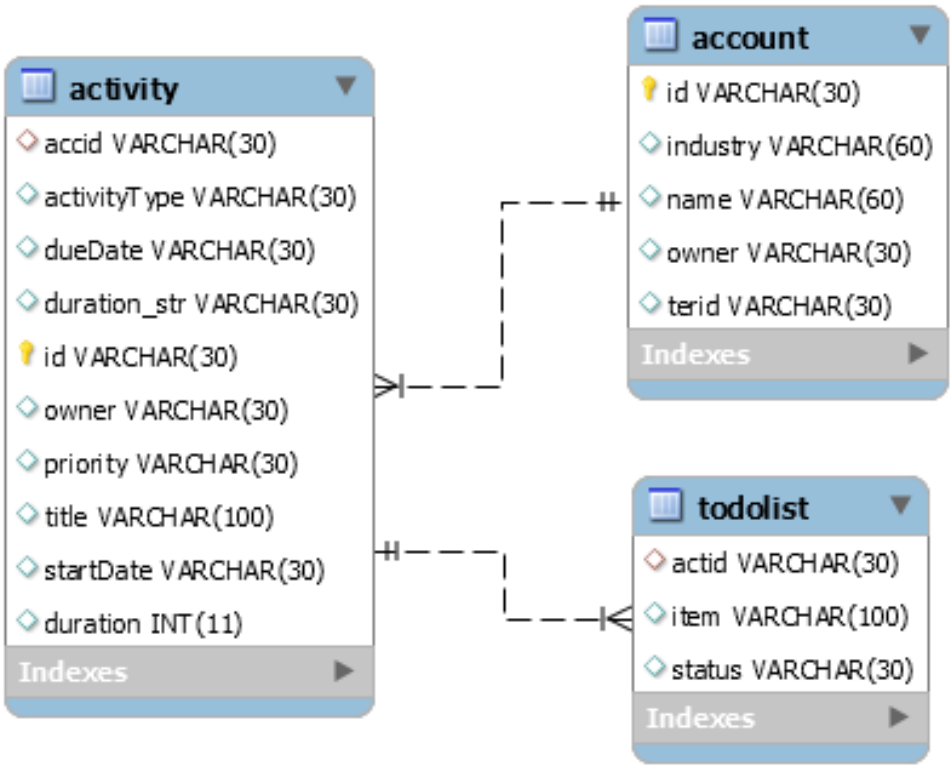


Figure 3.3.1 Relational model for the Sales activities scenario

Figure 3.3.2 demonstrates a JSON model for the current scenario.

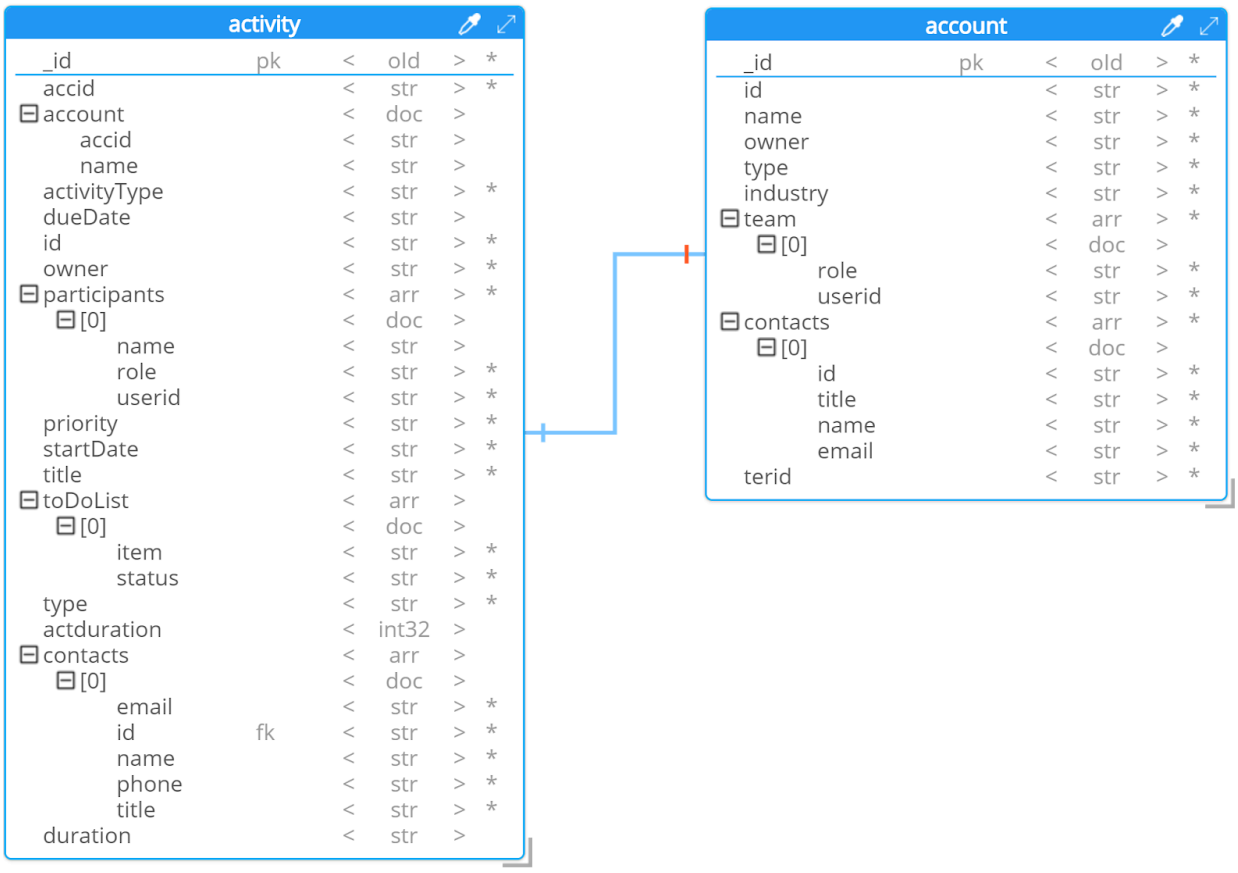


Figure 3.3.2 JSON model for the Sales activities scenario

The current scenario can be achieved with all three languages.

The query described in Listing 3.3.1 is an implementation of the current scenario for MySQL.

Listing 3.3.1 An SQL implementation for the Sales activities scenario

```
SELECT  ac.industry,
 SUM( CASE WHEN a.activitytype = 'Task' THEN 1 ELSE 0 END ) task,
 SUM( CASE WHEN a.activitytype = 'Appointment' THEN 1 ELSE 0 END ) appts
FROM crm.activity a
 INNER JOIN crm.account ac ON (a.accid = ac.id)
WHERE  a.startdate BETWEEN '2018-10-01' AND '2018-12-31'
GROUP BY ac.industry
```

The query described in Listing 3.3.2 is an implementation of the current scenario for N1QL.

Listing 3.3.2 An N1QL implementation for the Sales activities scenario

```
SELECT  ac.industry,
 SUM( CASE WHEN a.activityType = 'Task' THEN 1 ELSE 0 END) task,
 SUM( CASE WHEN a.activityType = 'Appointment' THEN 1 ELSE 0 END ) appts
FROM crm.activity a
```

```
 INNER JOIN crm.account ac ON a.accid = ac.id
WHERE a.startDate BETWEEN '2018-10-01' AND '2018-12-31'
GROUP BY  ac.industry
```

The query described in Listing 3.3.3 is an implementation of the current scenario for a MongoDB query.

Listing 3.3.3 A MongoDB query implementation for the Sales activities scenario

```
db.activity.aggregate(
  { $match: { startDate: { $gt: '2018-01-01', $lt: '2018-12-31' } } },
  {
     $lookup: {
         from: "account",
         localField: "accid",
         foreignField: "id",
         as: "account_docs"
     }
  },
  { $match: { "account_docs": { $ne: [] } } },
  { $unwind: "$account_docs" },
  {
     $project: {
       item: 1,
       task: { $cond: { if: { $eq: ["$activityType", "Task"] }, then: 1,
else: 0 } },
       appt: { $cond: { if: { $eq: ["$activityType", "Appointment"] },
then: 1, else: 0 } }
     }
   },
   {
     $group: {
       _id: "$account_docs.industry",
       tasks: { $sum: "$task" },
       appointments: { $sum: "$appt" }
     }
   }
);
```

Summary

This scenario consists of two main parts: aggregation functions and conditions. In aggregation, the function has to be a sum calculation. In conditions, two activity types should be included in the result: *Appointment* and *Task*.

Each database supports functionality for executing this scenario. MySQL and N1QL are similar and provide `SUM` as an aggregation function for our query and `CASE` for conditioning. A MongoDB query has operators for achieving the same results: `SUM` and `COND`. These things are similar to the MySQL functionalities, but the process of using these functions is different, because a MongoDB query is not an SQL-like language. Therefore, it should be noted that in this case, a MongoDB query is more complicated due to the nature of the language.

This scenario is not difficult to accomplish in each of the database languages. N1QL and MySQL are similar owing to N1QL being an SQL-like language. A MongoDB query achieves the same result as N1QL and MySQL. However, a MongoDB query has less expressiveness due to the nature of the language.

Table 3.3.1 Metrics for the Sales activities scenario

Criteria	MySQL	N1QL	MongoDB query
Simplicity	9	9	9
Readability	10	10	10
Expressiveness	9	9	8
Flexibility	9	9	9
Skills availability	9	7	7
A number of code lines	7	7	27
A number of client/server trips	1	1	1

3.4. Sales organizations

I run a sales organization for the *Aggressive Achievers* territory. I want to find out how much time we spent talking to the accounts assigned to this territory for *Q3FY19*.

Figure 3.4.1 demonstrates a relational model for the current scenario.

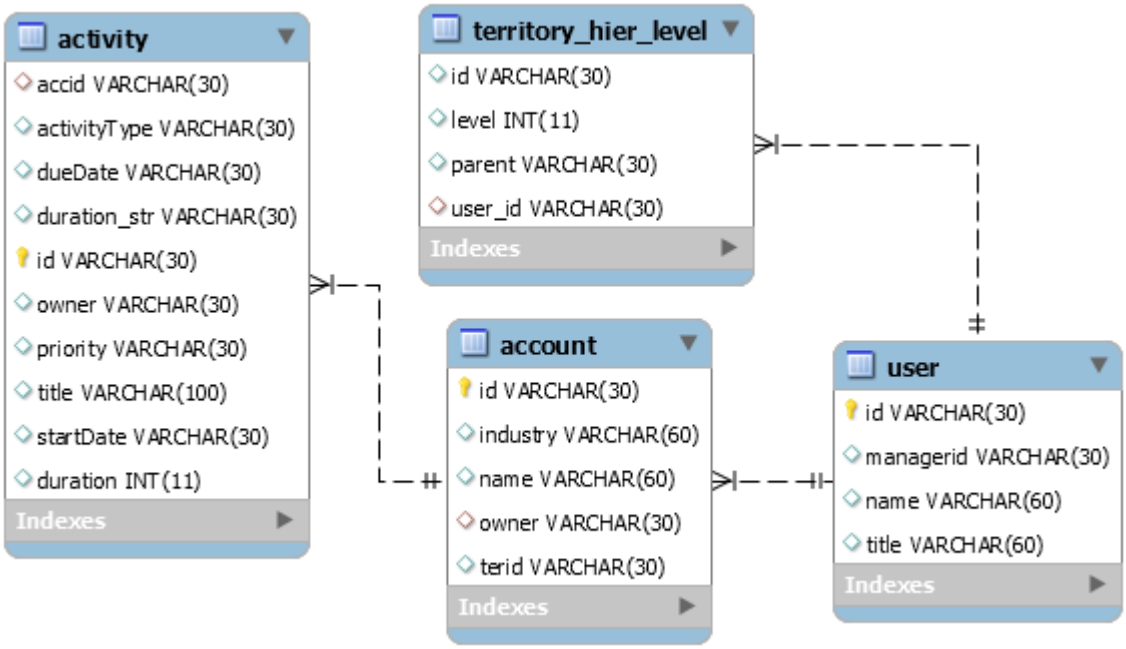


Figure 3.4.1 A relational model for the Sales organizations scenario

Figure 3.4.2 demonstrates a JSON model for the current scenario.

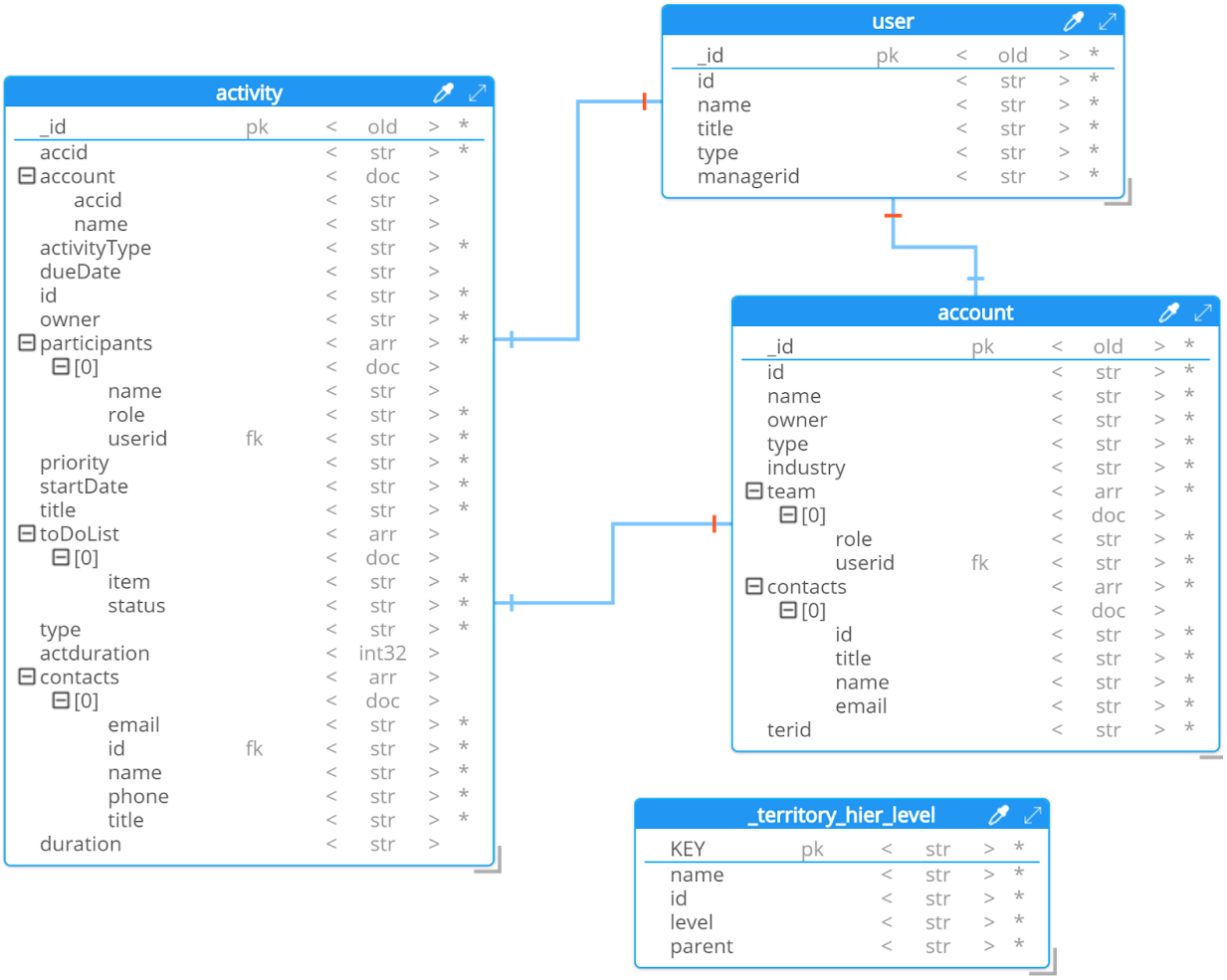


Figure 3.4.2 A JSON model for the Sales organizations scenario

The current scenario can be achieved with all three languages.

The query described in Listing 3.4.1 is an implementation of the current scenario for MySQL.

Listing 3.4.1 An SQL implementation for the Sales organizations scenario

```
SELECT cus.name Customer,
 cus.industry Industry,
 usr.name Owner,
 COUNT(1) NumOfMeetings,
 SUM(act.duration) `Time Spent`,
 ROUND( PERCENT_RANK()
   OVER (ORDER BY SUM(act.duration)),2) `PctRank`
FROM crm.activity act
 INNER JOIN crm.account cus ON act.accid = cus.id
 INNER JOIN crm.`user` usr ON cus.owner = usr.id
 INNER JOIN (
       SELECT thl.id
       FROM crm.territory_hier_level thl
```

```
        WHERE   hl.parent = 'ter3'
      ) ter ON cus.terid = ter.id
WHERE  act.activityType = 'Appointment'
 AND act.startDate BETWEEN '2018-10-01' AND '2018-12-31'
GROUP BY  cus.name,
 cus.industry,
 usr.name;
```

The query described in Listing 3.4.2 is an implementation of the current scenario for N1QL.

Listing 3.4.2 An N1QL implementation for the Sales organizations scenario

```
SELECT cus.name Customer,
 cus.industry Industry,
 usr.name Owner,
 COUNT(1) NumOfMeetings,
 SUM(TO_NUMBER(act.duration)) `Time Spent`,
 ROUND( PERCENT_RANK()
   OVER ( ORDER BY SUM(TO_NUMBER(act.duration)) ),2 ) `PctRank`
FROM crm act
 INNER JOIN crm cus ON (act.accid = cus.id AND cus.type = 'account')
 INNER JOIN crm usr ON( cus.owner = usr.id AND usr.type = 'user')
 INNER JOIN (
       SELECT thl.id
       FROM crm thl
       WHERE  thl.type = '_territory_hier_level'
        AND thl.parent = 'ter3'
     ) ter ON cus.terid = ter.id
WHERE act.type = 'activity'
 AND act.activityType = 'Appointment'
 AND act.startDate, 'month' BETWEEN '2018-10-01' AND '2018-12-31'
GROUP BY  cus.name,
 cus.industry,
 usr.name
```

The query described in Listing 3.4.3 is an implementation of the current scenario for a MongoDB query.

Listing 3.4.3 A MongoDB query implementation for the Sales organizations scenario

```
var territory_tier3 = db.territory.aggregate([
  {
     $graphLookup:
       {
         from: "territory",
         startWith: "$parentid",
         connectFromField: "parentid",
         connectToField: "id",
         as: "territoryHierarchy"
       }
  },
  { $unwind: "$territoryHierarchy" },
  { $match: { "territoryHierarchy.id": "ter3" } }
]).map(function (ter) {
```

```
    return ter.id;
})

var listOfTimeSpent = db.activity.aggregate([
  { $match: { "activityType": "Appointment" } },
  { $match: { "startDate": { $gt: '2018-10-01', $lt: '2018-12-31' } } },
  {
     $lookup: {
       from: "account",
       localField: "accid",
       foreignField: "id",
       as: "account_docs"
     }
  },
  { $match: { "account_docs": { $ne: [] } } },
  { $unwind: "$account_docs" },
  {
     $lookup: {
       from: "user",
       localField: "account_docs.owner",
       foreignField: "id",
       as: "ad_user"
     }
  },
  { $match: { "ad_user": { $ne: [] } } },
  { $match: { "account_docs.terid": { $in: territory_tier3 } } },
  {
     $group:
       {
         _id: {
           customer: "$account_docs.name",
           industry: "$account_docs.industry",
           owner: "$account_docs.owner"
         },
         numberOfMeetings: { $sum: 1 },
         timeSpent: { $sum: "$actduration" }
       }
    },
    { $sort: { timeSpent: 1 } },
    { $group: { _id: null, ts: { "$addToSet": "$timeSpent" } } },
    { $unwind: "$ts" },
    { $sort: { ts: 1 } }
]).map(function (t) {
   return t.ts;
})

db.activity.aggregate([
  { $match: { "activityType": "Appointment" } },
  { $match: { "startDate": { $gt: '2018-10-01', $lt: '2018-12-31' } } },
  {
     $lookup: {
         from: "account",
         localField: "accid",
         foreignField: "id",
```

```
            as: "account_docs"
         }
      },
      { $match: { "account_docs": { $ne: [] } } },
      { $unwind: "$account_docs" },
      {
         $lookup: {
             from: "user",
             localField: "account_docs.owner",
             foreignField: "id",
             as: "ad_user"
          }
       },
      { $match: { "ad_user": { $ne: [] } } },
      { $match: { "account_docs.terid": { $in: territory_tier3 } } },
      {
         $group:
           {
             _id: {
                customer: "$account_docs.name",
                industry: "$account_docs.industry",
                owner: "$account_docs.owner"
             },
             numberOfMeetings: { $sum: 1 },
             timeSpent: { $sum: "$actduration" }
         }
      },
      { $addFields: { pctRank: { $indexOfArray: [listOfTimeSpent,
"$timeSpent"] } } },
      { $addFields: { pctRank: { $divide: ["$pctRank",
listOfTimeSpent.length] } } },
      {
         $addFields:
           { pctRank:
              { '$divide': [{ '$trunc': { '$add': [{ '$multiply':
['$pctRank', 100] }, 0.5] } }, 100] } }
      }
])
```

Summary

In this scenario, two functionalities—hierarchy and window—should be noted.

The chosen territory is *ter3*. This means the result should only include territories where the parent is *ter3*. For MySQL and N1QL, a special table is created for storing territory hierarchy. However, a MongoDB query does not require this extra table as `$graphLookup` can be used for building a hierarchy within the query.

Another important point in this scenario is a percent rank. It is a window function, which calculates percentile. N1QL and MySQL support it by default with `percent_rank()`. A MongoDB query does not support it, so another method should be chosen as a replacement. Subqueries using *var* are suitable in this case. Unfortunately, this leads to duplications in the query.

This scenario is achievable for each database. MySQL and N1QL have good readability and less complexity in comparison to a MongoDB query. In its turn, a MongoDB query has a lot of duplication in the query, because it does not support percentile calculation. Despite the fact that a MongoDB query is complex, it does not need to support an extra table for territory hierarchy. It is also worth mentioning that N1QL and MySQL can achieve the desired result with a single query, but a MongoDB query cannot achieve the same result using a single query due to functional limitations.

Table 3.4.1 Metrics for the Sales organizations scenario

Criteria	MySQL	N1QL	MongoDB query
Simplicity	8	8	2
Readability	9	9	1
Expressiveness	9	9	1
Flexibility	9	9	4
Skills availability	6	6	1
A number of code lines	20	22	103
A number of client/server trips	1	1	3

3.5. A sales task report

This scenario shows how the number of sales-related tasks have changed a month over a month during the year of 2018.

Figure 3.5.1 demonstrates a relational model for the current scenario.

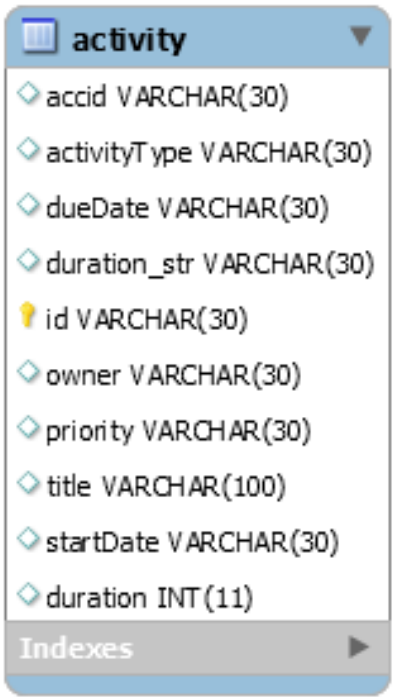


Figure 3.5.1 A relational model for the Sales task report scenario

Figure 3.5.2 demonstrates a JSON model for the current scenario.

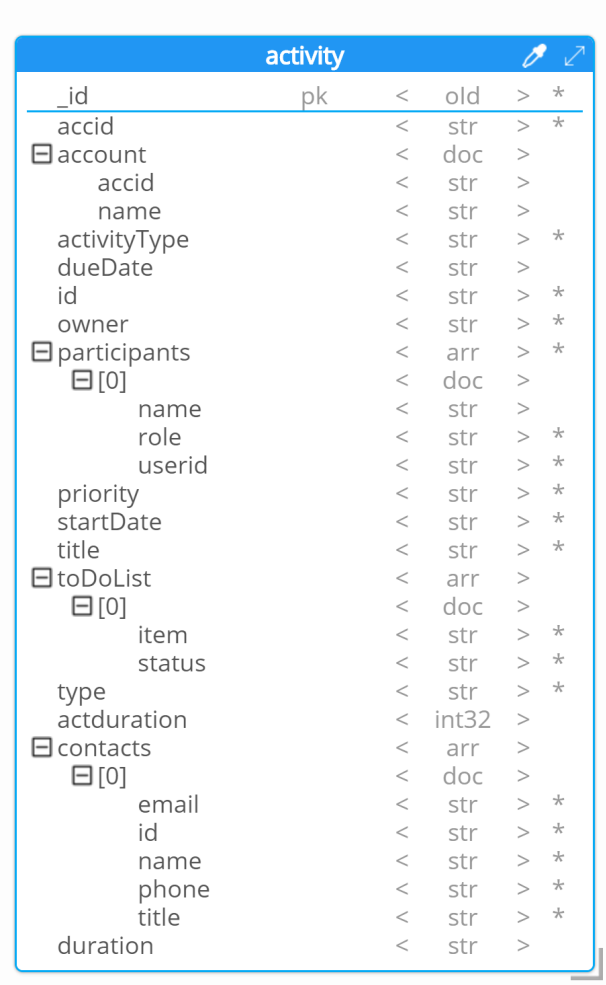


Figure 3.5.2 A JSON model for the Sales task report scenario

The current scenario can be achieved with all three languages.

The query described in Listing 3.5.1 is an implementation of the current scenario for MySQL.

Listing 3.5.1 An SQL implementation for the Sales task report scenario

```
WITH current_period_task AS (
 SELECT  DATE_FORMAT(a.startDate, '%Y-%m')  month,
  COUNT(1)  current_period_task_count
 FROM crm.activity a
 WHERE a.activityType = 'Task'
  AND DATE_FORMAT(a.startDate, '%Y') = 2018
 GROUP BY DATE_FORMAT(a.startDate, '%Y-%m')
),
last_period_task AS (
  SELECT  x.month,
   x.current_period_task_count,
   LAG(x.current_period_task_count)
     OVER ( ORDER BY x.month )  last_period_task_count
```

```
  FROM current_period_task x
)
SELECT b.month,
 b.current_period_task_count,
 ROUND( ( ( b.current_period_task_count - b.last_period_task_count
          ) / b.last_period_task_count), 2) MoMChg
FROM last_period_task AS b;
```

The query described in Listing 3.5.2 is an implementation of the current scenario for N1QL.

Listing 3.5.2 An N1QL implementation for the Sales task report scenario

```
WITH current_period_task AS (
 SELECT DATE_TRUNC_STR(a.startDate, 'month') month,
  COUNT(1)  current_period_task_count
 FROM crm a
 WHERE a.type = 'activity'
  AND a.activityType = 'Task'
  AND DATE_PART_STR(a.startDate, 'year') = 2018
 GROUP BY DATE_TRUNC_STR(a.startDate, 'month')
),
last_period_task AS (
  SELECT  x.month,
    x.current_period_task_count,
    LAG(x.current_period_task_count)
      OVER ( ORDER BY x.month )  last_period_task_count
  FROM
   current_period_task x
)
SELECT b.month,
 b.current_period_task_count,
 ROUND( ( (b.current_period_task_count - b.last_period_task_count
         ) / b.last_period_task_count), 2 )  MoMChg
FROM last_period_task AS b
```

The query described in Listing 3.5.3 is an implementation of the current scenario for a MongoDB query.

Listing 3.5.3 A MongoDB query implementation for the Sales task report scenario

```
var res = db.activity.aggregate([
  {
     $project: {
       activityType: 1,
       month: { $month: { $dateFromString: { dateString: "$startDate" } } }
},
       year: { $year: { $dateFromString: { dateString: "$startDate" } } }
      }
  },
  { $match: { $and: [{ "activityType": "Task" }, { "year": { $eq: 2018 } }]
} },
  {
     $group: {
       _id: { $dateFromParts: { year: "$year", month: "$month" } },
```

```
        count: { $sum: 1 }
      }
    },
    { $sort: { _id: 1 } }
]).map(function (el) { return el.count });

db.activity.aggregate([
  {
    $project: {
      activityType: 1,
      month: { $month: { $dateFromString: { dateString: "$startDate" } } },
      year: { $year: { $dateFromString: { dateString: "$startDate" } } }
    }
  },
  {
    $match: {
        $and: [
            { "activityType": "Task" },
            { "year": { $eq: 2018 } }
        ]
    }
  },
  {
    $group: {
      _id: { $dateFromParts: { year: "$year", month: "$month" } },
      count: { $sum: 1 }
    }
  },
  {
    $project: {
      _id: 1,
      count: 1,
      MoMChg: {
        $cond: [
         { $ne: [{ $month: "$_id" }, 1] },
         {
            $divide: [
              { $subtract: ["$count", { $arrayElemAt: [res, { $subtract: [{
$month: "$_id" }, 2] }] }] },
              { $arrayElemAt: [res, { $subtract: [{ $month: "$_id" }, 2] }]
}]
            }, null]
          }
      }
    },
    { $sort: { _id: 1 } },
    {
       $project: {
         _id: 0,
         month: { $concat: [{ $toString: { $year: "$_id" } }, "-", {
$toString: { $month: "$_id" } }] },
         current_period_task_count: "$count",
         MoMChg: {
           '$divide': [
```

```
                { '$trunc':
                    { '$add':
                        [{ '$multiply': ['$MoMChg', 100] }, 0.5]
                        }
                }, 100]
            }
        }
    }
```

Summary

In this scenario, the first function in the query sets a date from a string. This is needed in order to filter results by year and to combine by month. A MongoDB query has the most complex way since the *String* should be converted to *Date*, and then the month and year can be extracted. Other query languages provide seamless ways to work with the date in a usual format.

N1QL and MySQL support the `WITH` operator to store intermediate request results. On the other hand, a MongoDB query uses variables, which can be initiated with intermediate values. Operations like grouping and ordering have the same complexity and readability for all queries.

The `LAG` function is used to compare the values for each month with the previous one. Both N1QL and MySQL support window functions, making it simple to achieve this comparison. The only way to achieve this result in a MongoDB query is by creating an additional request and using it in the main part of query.

The next important point is arithmetic operations. N1QL and MySQL have the most readability in this case, while Mongo is more complex. The last step is rounding to two decimal places. MySQL and N1QL provide the `ROUND` function. Mongo does not support rounding, so some workaround should be applied.

MySQL and N1QL queries look pretty similar. Both support window functions that make the query easier to read and support. MongoDB query looks more complex and uses some workarounds to achieve the same result but with a lower readability. The method of applying arithmetic operations and the difficulties during roundings seem to be the biggest drawbacks in this scenario.

Table 3.5.1 Metrics for the Sales task report scenario

Criteria	MySQL	N1QL	MongoDB query
Simplicity	9	9	7
Readability	8	8	7
Expressiveness	9	9	8
Flexibility	9	9	7
Skills availability	9	1	2
A number of code lines	20	22	71
A number of client/server trips	1	1	2

3.6. A skill set report

The company is performing an analysis on the sales team skill sets/roles in the current sales organization. It needs to identify all the territories where there is only a single person with a specific role/skill set and that the territory handles more than five accounts.

Figure 3.6.1 demonstrates a JSON model for the current scenario.

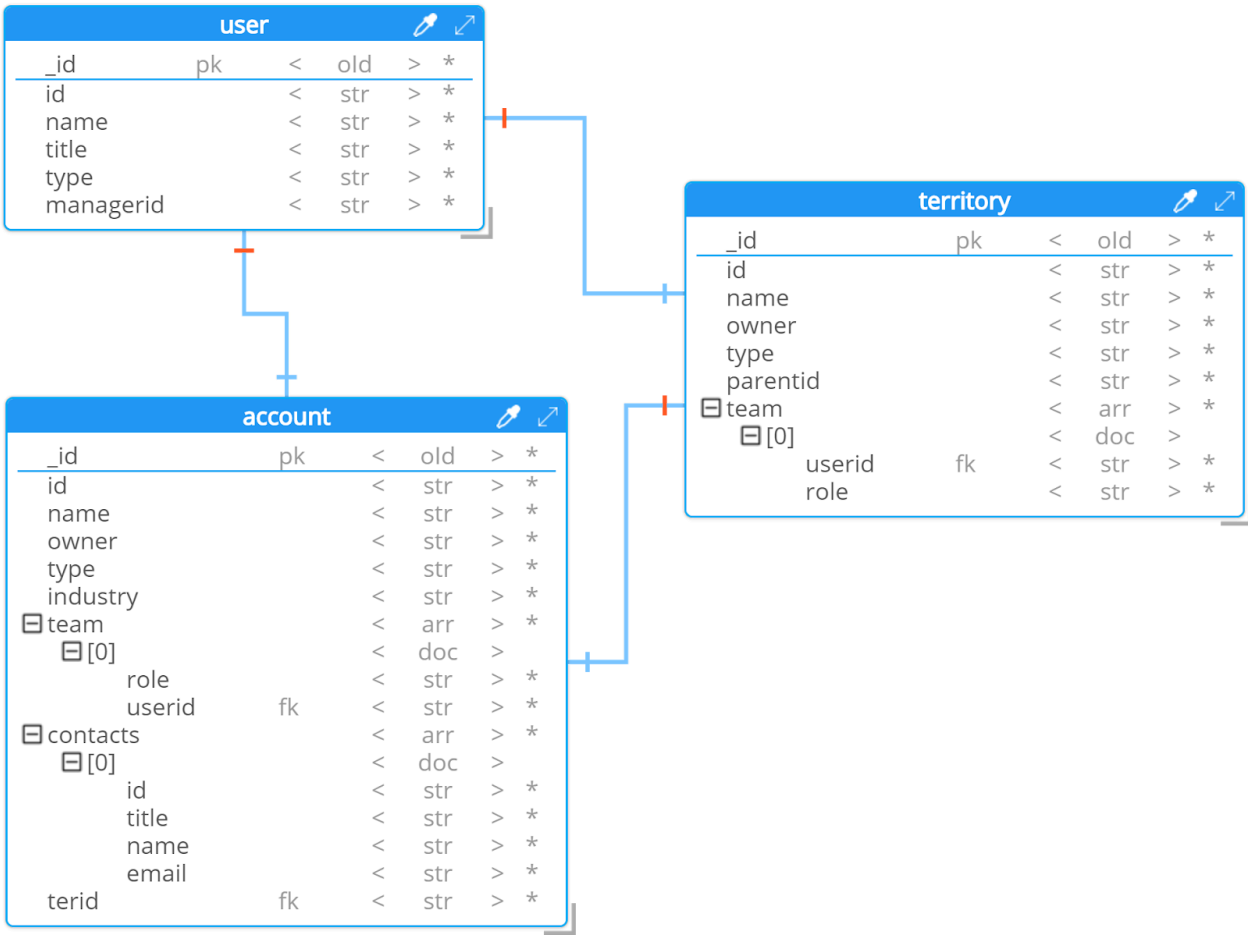


Figure 3.6.1 A JSON model for the Skill set report scenario

The current scenario can be achieved with N1QL and a MongoDB query.

The query described in Listing 3.6.1 is an implementation of the current scenario for N1QL.

Listing 3.6.1 An N1QL implementation for the Skill set report scenario

```
SELECT tar.territory,
 rroles
FROM
 (
   SELECT terrs.territory,
     SUM(custcount) numofcusts,
     ARRAY_FLATTEN(ARRAY_AGG(terrs.combined_roles), 1) allroles
    FROM
```

```
     (
       SELECT    t.name territory,
         COUNT(c.id) custcount,
         ARRAY_CONCAT(c.team [*].`role`, t.team [*].`role`) combined_roles
       FROM account c
         INNER JOIN territory t ON (c.terid = t.id AND t.type =
'territory')
       WHERE c.type = 'account'
       GROUP BY t.name,
                c.team [*].`role`,
                t.team [*].`role`
      ) terrs
     GROUP BY terrs.territory
     HAVING SUM(custcount) >= 5
 )  tar
 UNNEST tar.allroles rroles
GROUP BY tar.territory,
         rroles
HAVING  COUNT(1) = 1
ORDER BY  tar.territory
```

The query described in Listing 3.6.2 is an implementation of the current scenario for a MongoDB query.

Listing 3.6.2 A MongoDB query implementation for the Skill set report scenario

```
db.account.aggregate([
  { $match: { "type": "account" } },
  {
     $lookup: {
       from: "territory",
       localField: "terid",
       foreignField: "id",
       as: "territories"
     }
  },
  { $match: { "territories": { $ne: [] } } },
  { $project: { id: 1, team: 1, territory: { "$arrayElemAt":
["$territories", 0] } } },
  { $match: { "territory.type": "territory" } },
  {
     $project: {
       id: 1,
       territory: "$territory.name",
       team: { $map: { input: "$team", as: "member", in: "$$member.role" }
},
       territoryTeam: { $map: { input: "$territory.team", as: "member", in:
"$$member.role" } }
     }
 },
 {
  $group: {
    _id: {
        territory: "$territory",
```

```
        accountTeam: "$team",
        territoryTeam: "$territoryTeam"
      },
      count: { $sum: 1 }
    }
  },
  {
    $project: {
      _id: {
        territory: "$_id.territory",
        combined_roles: { "$concatArrays": ["$_id.accountTeam",
"$_id.territoryTeam"] }
      },
      count: 1
    }
  },
  {
    $group: {
      _id: { territory: "$_id.territory", },
      allroles: { $push: "$_id.combined_roles" },
      custcount: { $sum: "$count" }
         }
    },
    { $match: { custcount: { $gte: 5 } } },
    {
       $project: {
         _id: 1,
         allroles: {
            $reduce:
               { input: "$allroles", initialValue: [], in: { $concatArrays:
["$$value", "$$this"] } }
         }
       }
    },
    { $unwind: "$allroles" },
    {
       $group: {
         _id: {
            territory: "$_id.territory",
            rroles: "$allroles"
         },
         count: { $sum: 1 }
       }
    },
    { $match: { count: { $eq: 1 } } },
    {
       $project: {
           _id: 0,
           territory: "$_id.territory",
           rroles: "$_id.rroles"
       }
    },
```

```
{ $sort: { "territory": 1 } }])
```

Summary

The intermediate goal of this query is to combine account team members with territories by setting conditions on the type of account and the type of territory.

While getting the results, arrays that contains information about team members should be concatenated. The MongoDB query and N1QL approaches are similar. Both have readable methods for uniting arrays.

Another array operation that is used here is flattening the structure. N1QL provides `ARRAY_FLATTEN`, while Mongo uses reduce operation. This approach is not as simple as `ARRAY_FLATTEN` but it achieves the same result.

To expand results for each role N1QL uses `UNNEST` and Mongo uses the `UNWIND` operation. In this context, both look similar and accomplish the same result.
Mongo and N1QL use different approaches to achieve the same result. It is difficult to compare which is more suitable for this scenario, but readability is higher with N1QL. It is simpler to explain to Couchbase which data should be returned. MongoDB query is longer, but it shows every step in detail.

Table 3.6.1 Metrics for the Skill set report scenario

Criteria	MySQL (unsupported)	N1QL	MongoDB query
Simplicity	-	9	8
Readability	-	9	8
Expressiveness	-	9	8
Flexibility	-	9	9
Skills availability	-	7	7
A number of code lines	-	27	76
A number of client/server trips	-	1	1

3.7. Search contacts

A query to review all the presentations that we have conducted with customers in *CY19Q4*. The query needs to show the time we spent for each meeting, the running count, and the percentage of time for the meeting over the total time we spent talking to the customer. Furthermore, the query needs to calculate a `high_touch_rank`, which is the percentage of the customer's contacts who attended the meeting against the total number of the customer's contacts.

Figure 3.7.1 demonstrates a JSON model for the current scenario.

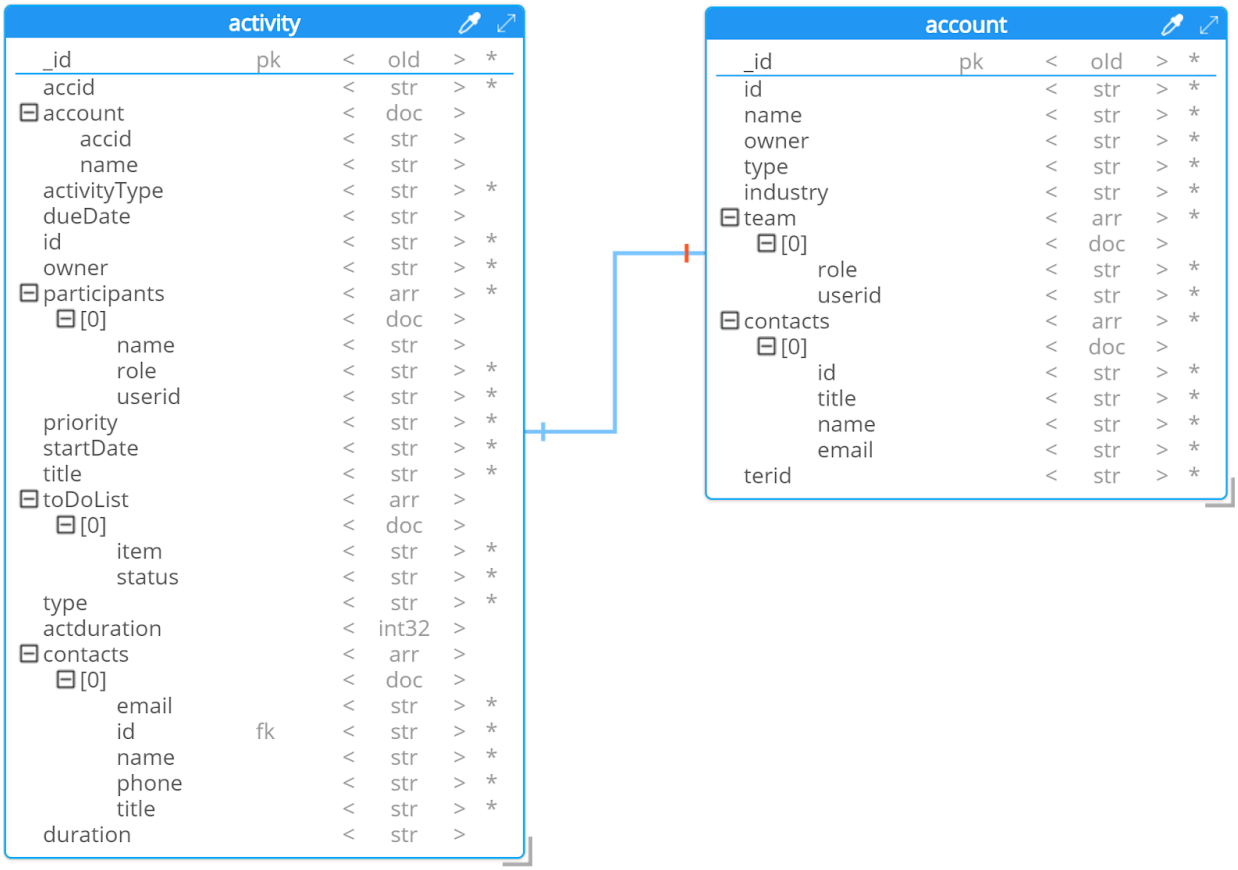


Figure 3.7.1 A JSON model for the Search contacts scenario

The current scenario can be achieved with N1QL and a MongoDB query.

The query described in Listing 3.7.1 is an implementation of the current scenario for N1QL.

Listing 3.7.1 An N1QL implementation for the Search contacts scenario

```
SELECT c.name,
 a.title,
 a.actduration,
 a.startDate,
 SUM(a.actduration)
   OVER ( PARTITION BY c.name ORDER BY c.name, a.startDate ) running_total,
 TRUNC(100*(a.actduration/ SUM(a.actduration)
                           OVER(PARTITION BY c.name)) ) pct_of_total_time,
```

```
 RANK() OVER (PARTITION BY c.name ORDER BY
    (ARRAY_COUNT(a.contacts) / ARRAY_COUNT(c.contacts))DESC) hightouch_rank
FROM crm a
 INNER JOIN crm c ON (a.accid = c.id AND c.type = 'account')
WHERE a.type = 'activity'
 AND a.activityType = 'Appointment'
 AND a.startDate BETWEEN '2018-10' AND '2018-12'
GROUP BY c.name,
  a.title,
  a.startDate,
  a.actduration,
  a.contacts,
  c.contacts
ORDER BY c.name,
 a.startDate
```

The query described in Listing 3.7.2 is an implementation of the current scenario for a MongoDB query.

Listing 3.7.2 A MongoDB query implementation for the Search contacts scenario

```
var accountContacts = db.activity.aggregate([
  { $match: { type: "activity" } },
  { $match: { activityType: "Appointment" } },
  { $match: { startDate: { $gt: '2018-10-01', $lt: '2018-12-31' } } },
  {
     $lookup:
       {
         from: "account",
         localField: "accid",
         foreignField: "id",
         as: "account_docs"
     }
   },
   { $match: { "account_docs": { $ne: [] } } },
   { $unwind: "$account_docs" },
   {
      $group: {
        "_id": { name: "$account_docs.name" },
        activity_contacts: { $addToSet: "$contacts" },
        account_contacts: { $addToSet: "$account_docs.contacts" },
      }
    },
    { $unwind: "$account_contacts" },
    {
       $project: {
         _id: 0,
         name: "$_id.name",
         sizeAccountContacts: { $size: "$account_contacts" }
       }
    }
]).map(function (total) {
   return total;
})
```

```
var rank_temp = db.activity.aggregate([
  { $match: { type: "activity" } },
  { $match: { activityType: "Appointment" } },
  { $match: { startDate: { $gt: '2018-10-01', $lt: '2018-12-31' } } },
  {
     $lookup:
       {
         from: "account",
         localField: "accid",
         foreignField: "id",
         as: "account_docs"
       }
  },
  { $match: { "account_docs": { $ne: [] } } },
  { $unwind: "$account_docs" },
  {
     $group: {
       "_id": {
         name: "$account_docs.name",
         title: "$title",
         startDate: "$startDate",
         duration: "$actduration",
         activity_contacts: "$contacts"
       }
      }
    },
    { $addFields: { accountContacts: accountContacts } },
    {
       $project: {
         "_id": 0,
         name: "$_id.name",
         title: "$_id.title",
         startDate: "$_id.startDate",
         duration: "$_id.duration",
         accountName: "$_id.name",
         sizeCurrentActivityContacts: { $size: "$_id.activity_contacts" },
         sizeAccountContacts: {
            $filter: {
               input: "$accountContacts",
               as: "element",
               cond: { $eq: ["$$element.name", "$_id.name"] }
            }
          }
        }
    },
    {
        $unwind: "$sizeAccountContacts"
    },
    { $addFields:{ sizeAccountContacts:
"$sizeAccountContacts.sizeAccountContacts" } },
    { $addFields: {
       hightouch_rank: {
       '$divide': ["$sizeCurrentActivityContacts", "$sizeAccountContacts"]
```

```
        }
      }
    },
    { $addFields: { hightouch_rank: { '$multiply': ["$hightouch_rank", 10]
} } },
    { $addFields: { hightouch_rank: { $trunc: "$hightouch_rank" } } },
]).map(function (total) {
    return total;
})

var total_time = db.activity.aggregate([
  { $match: { type: "activity" } },
  { $match: { activityType: "Appointment" } },
  { $match: { startDate: { $gt: '2018-10-01', $lt: '2018-12-31' } } },
  {
     $lookup:
       {
        from: "account",
        localField: "accid",
        foreignField: "id",
        as: "account_docs"
       }
    },
    { $match: { "account_docs": { $ne: [] } } },
    { $match: { "account_docs.type": "account" } },
    { $unwind: "$account_docs" },
    {
       $group: {
         _id: { name: "$account_docs.name" },
         total_time: { $sum: "$actduration" }
       }
    },
    {
      $project: {
        "_id": 0,
        name: "$_id.name",
        total_time: "$total_time"
      }
    }
]).map(function (total) {
    return total;
})

var times = db.activity.aggregate([
  { $match: { type: "activity" } },
  { $match: { activityType: "Appointment" } },
  { $match: { startDate: { $gt: '2018-10-01', $lt: '2018-12-31' } } },
  {
     $lookup:
        {
           from: "account",
           localField: "accid",
           oreignField: "id",
           as: "account_docs"
```

```
            }
        },
        { $match: { "account_docs": { $ne: [] } } },
        { $match: { "account_docs.type": "account" } },
        { $unwind: "$account_docs" },
        {
            $sort: { startDate: 1 }
        },
        {
            $group: {
                "_id": { name: "$account_docs.name" },
                time: { $push: "$actduration" }
            }
        },
        {
            $project: {
                "_id": 0,
                name: "$_id.name",
                time: "$time"

            }
        },
        {
            $sort: { name: 1 }
        }
]).map(function (total) {
    return total;
})

var running_total = db.activity.aggregate([
    { $match: { type: "activity" } },
    { $match: { activityType: "Appointment" } },
    { $match: { startDate: { $gt: '2018-10-01', $lt: '2018-12-31' } } },
    {
        $lookup:
            {
                from: "account",
                localField: "accid",
                foreignField: "id",
                as: "account_docs"
            }
    },
    { $match: { "account_docs": { $ne: [] } } },
    { $match: { "account_docs.type": "account" } },
    { $unwind: "$account_docs" },
    { $sort: { "name": 1, "startDate": 1 } },
    {
        $group: {
            "_id": { name: "$account_docs.name" },
            "items": { "$push": "$$ROOT" }
        }
    },
    { $unwind: { "path": "$items", "includeArrayIndex": "items.rank" } },
    { $replaceRoot: { "newRoot": "$items" } },
```

```
    { $sort: { "name": 1, "startDate": 1 } },
    { $addFields: { times: times } },
    {
        $project: {
            name: "$account_docs.name",
            title: "$title",
            startDate: "$startDate",
            rank: "$rank",
            times: {
                $filter: {
                    input: "$times",
                    as: "element",
                    cond: { $eq: ["$$element.name", "$account_docs.name"] }
                }
            }
        }
    },
    { $unwind: "$times" },
    {
        $addFields: {
            rank: {
                $sum: ["$rank", 1]
            }
        }
    },
    {
        $addFields: {
            times: {
                $slice: ["$times.time", "$rank"]
            }
        }
    },
    {
        $addFields: {
            times: {
                $reduce: {
                    input: "$times",
                    initialValue: 0,
                    in: { $sum: ["$$value", "$$this"] }
                }
            }
        }
    },
    {
        $project: {
            _id: 0,
            name: "$name",
            title: "$title",
            startDate: "$startDate",
            times: "$times"
        }
    },
    { $sort: { "name": 1, "startDate": 1 } }
]).map(function (total) {
```

```
    return total;
})

db.activity.aggregate([
    { $match: { type: "activity" } },
    { $match: { activityType: "Appointment" } },
    { $match: { startDate: { $gt: '2018-10-01', $lt: '2018-12-31' } } },
    {
        $lookup:
            {
                from: "account",
                localField: "accid",
                foreignField: "id",
                as: "account_docs"
            }
    },
    { $match: { "account_docs": { $ne: [] } } },
    { $unwind: "$account_docs" },
    {
        $group: {
            "_id": {
                name: "$account_docs.name",
                title: "$title",
                startDate: "$startDate",
                duration: "$actduration",
                activity_contacts: "$contacts",
                account_contacts: "$account_docs.contacts",
            }
        }
    },
    { $addFields: { total_time: total_time } },
    { $addFields: { hightouch_rank: rank_temp } },
    { $addFields: { running_total: running_total } },
    {
        $project: {
            "_id": 0,
            name: "$_id.name",
            title: "$_id.title",
            startDate: "$_id.startDate",
            duration: "$_id.duration",
            activity_contacts: "$_id.activity_contacts",
            account_contacts: "$_id.account_contacts",
            running_total: {
                $filter: {
                    input: "$running_total",
                    as: "element",
                    cond: {
                        $and: [
                            { $eq: ["$$element.name", "$_id.name"] },
                            { $eq: ["$$element.title", "$_id.title"] },
                            { $eq: ["$$element.startDate",
"$_id.startDate"] }
                        ]
                    }
```

```
                }
            },
            total_time: {
                $filter: {
                    input: "$total_time",
                    as: "element",
                    cond: { $eq: ["$$element.name", "$_id.name"] }
                }
            },
            hightouch_rank: {
                $filter: {
                    input: "$hightouch_rank",
                    as: "element",
                    cond: {
                        $and: [
                            { $eq: ["$$element.name", "$_id.name"] },
                            { $eq: ["$$element.title", "$_id.title"] },
                            { $eq: ["$$element.startDate",
"$_id.startDate"] }
                        ]
                    }
                }
            },
            sizeCurrentActivityContacts: { $size: "$_id.activity_contacts"
},
        }
    },
    { $unwind: "$total_time" },
    { $addFields: { pct_of_total_time: { '$divide': ["$duration",
"$total_time.total_time"] } } },
    { $addFields: { pct_of_total_time: { '$multiply':
["$pct_of_total_time", 100] } } },
    { $addFields: { pct_of_total_time: { $trunc: "$pct_of_total_time" } }
},
    { $unwind: "$hightouch_rank" },
    { $addFields: { hightouch_rank: "$hightouch_rank.hightouch_rank" } },
    {
        $project: {
            name: "$name",
            title: "$title",
            startDate: "$startDate",
            duration: "$duration",
            pct_of_total_time: "$pct_of_total_time",
            hightouch_rank: "$hightouch_rank",
            running_total: "$running_total.times"
        }
    },
    {
        $unwind: "$running_total"
    },
    { $sort: { name: 1, startDate: 1, hightouch_rank: -1 } }
])
```

Summary

In this scenario, we need to use the data from two collections and make analytical calculations.

In N1QL, `SUM` is used alongside `PARTITION` to get the time spent on each meeting. For a MongoDB query, the only way to archive this data is by creating a separate query.

With the `TRUNC` function of N1QL and other arithmetic operations, we can calculate the percentage of time for each meeting against the total time spent talking to the customer. A MongoDB query supports a similar method with the `$trunc` and `$addFields` functions.

To get the percentage of people who attended the meeting against all the customer's contacts, N1QL uses `RANK` over `PARTITION`. In case of MongoDB, it is needed to have additional queries.

This scenario shows the drawbacks of a MongoDB query in terms of working with analytics reports. To get the necessary results, developers should use additional queries. This makes the query complex and difficult to support. On the other hand, N1QL has a number of functions that help with making such reports simple and more readable. It also takes less code and time to achieve the same results.

MySQL cannot be compared in this scenario due to the lack of support for the array function.

Table 3.7.1 Metrics for the Search contacts scenario

Criteria	MySQL (unsupported)	N1QL	MongoDB query
Simplicity	-	8	1
Readability	-	8	1
Expressiveness	-	8	1
Flexibility	-	8	3
Skills availability	-	6	2
A number of code lines	-	23	347
A number of client/server trips	-	1	5

3.8. Calling Google Natural Language API

In order to find a hotel based on the most positive reviews, you can read through all the reviews for all the hotels or leverage Google Natural Language API to analyze the sentiment of the reviews. The query should return top 10 hotel reviews based on the sentiment score.

Figure 3.8.1 demonstrates a JSON model for the current scenario.

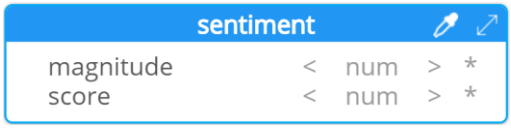


Figure 3.8.1 A JSON model for the Calling Google Natural Language API scenario

The current scenario can be achieved with N1QL only.

The query described in Listing 3.8.1 is an implementation of the current scenario for N1QL.

Listing 3.8.1 An N1QL implementation for the Calling Google Natural Language API scenario

```
SELECT ginfo.name,
 ginfo.review,
 ginfo.sentscore.documentSentiment.magnitude,
 ginfo.sentscore.documentSentiment.score
FROM
 (
  SELECT h.name,
   r.content  review,
   CURL( "https://language.googleapis.com/v1/documents:analyzeSentiment?
            key=YOUR_API_KEY_HERE",
      { "request": "POST",
        "header" :"Content-Type: application/json",
        "data": mydata }
    )  sentscore
  FROM `travel-sample` h
    UNNEST h.reviews r
    LET mydata = '{ "encodingType": "UTF8", "document": { "type":
"PLAIN_TEXT",
                  "content":"' || r.content || '"} }'
  WHERE  h.city = 'Nice'
) ginfo
ORDER BY ginfo.sentscore.documentSentiment.score DESC
LIMIT  10
```

Summary

N1QL offers specific functionality, such as calling third-party services via the REST API from a query. In our example, the query sends a request to Google Natural Language API for data analysis. Next, N1QL uses the response from the request to create a general table, which increases the flexibility of subsequent data processing. This functionality is appealing, because it removes the need to integrate extra services for loading data from third-party systems.

A MongoDB query and MySQL cannot implement this query, because they do not provide functionality for calling the API natively. N1QL provides flexibility and simplicity of integration with third-party systems, increasing our capability for analyzing data.

MySQL and a MongoDB query cannot be compared in this scenario, as they do not support the REST API in the query.

Table 3.8.1 Metrics for the Calling Google Natural Language API scenario

Criteria	MySQL (unsupported)	N1QL	MongoDB query (unsupported)
Simplicity	-	9	-
Readability	-	10	-
Expressiveness	-	9	-
Flexibility	-	9	-
Skills availability	-	6	-
A number of code lines	-	22	-
A number of client/server trips	-	1	-

3.9. Search criteria

Our goal is to identify the customer accounts and their related contacts where a particular topic has been discussed. The search criteria may include the following information partially or in full: a meeting title, a meeting date range, customer contact details, sales team member details (participants), and a customer name.

Figure 3.9.1 demonstrates a relational model for the current scenario.

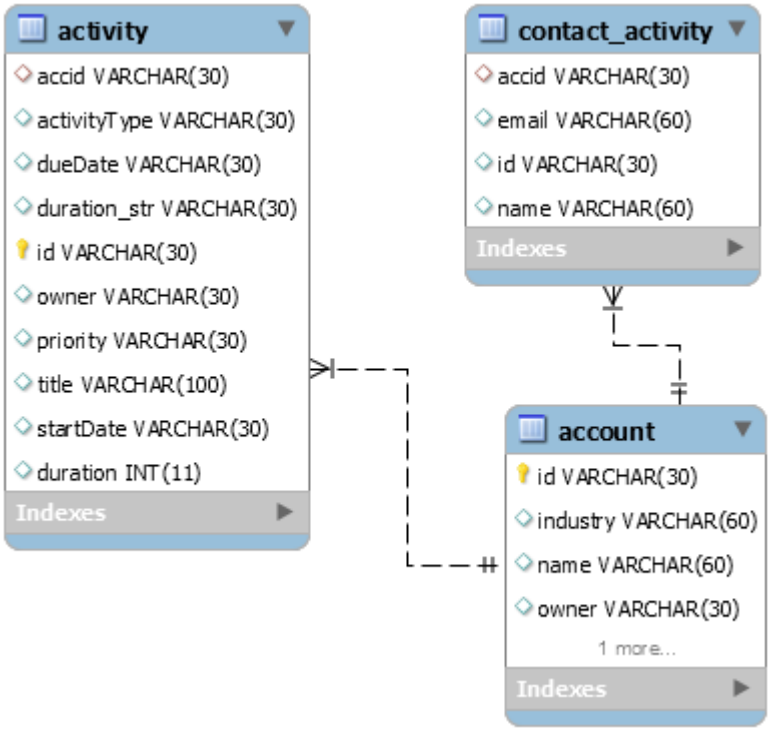


Figure 3.9.1 A relational model for the Search criteria scenario

Figure 3.9.2 demonstrates a JSON model for the current scenario.

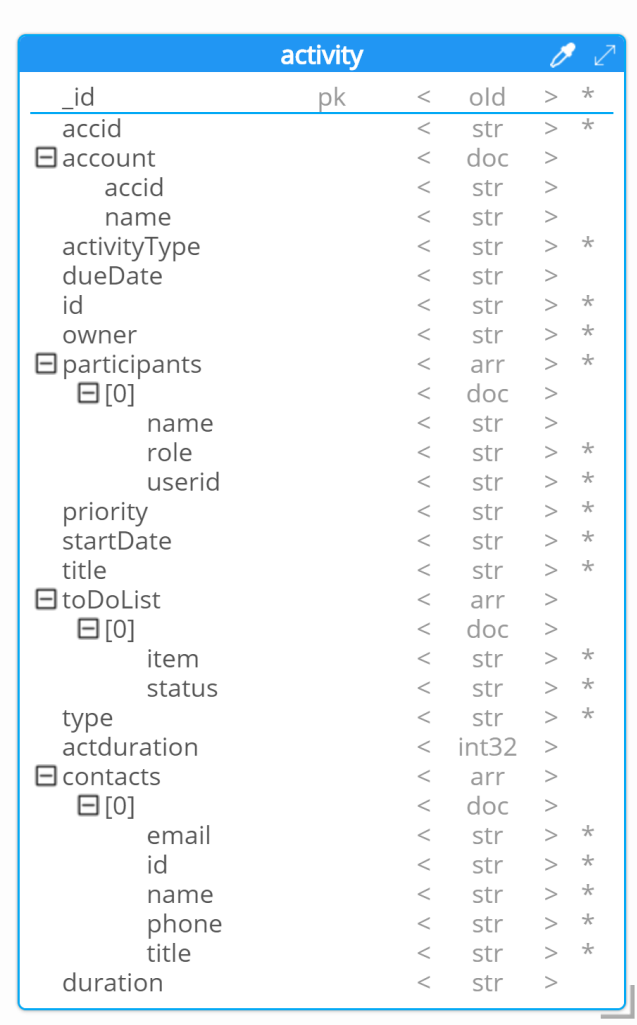


Figure 3.9.2 A JSON model for the Search criteria scenario

The current scenario can be achieved with all the three languages.

The query described in Listing 3.9.1 is an implementation of the current scenario for MySQL.

Listing 3.9.1 An SQL implementation for the Search criteria scenario

```
SELECT a.id,
 a.title,
 c.name customer,
 a.startDate,
 cn.name, cn.email
FROM activity a
 INNER JOIN account c ON a.accid = c.id
 INNER JOIN contact_activity cn ON c.id = cn.accid
WHERE a.activityType='Appointment'
 AND a.startDate BETWEEN '2016-08-29' AND '2016-08-30'
 AND UPPER(a.title) LIKE '%ARTIFICIAL INTELLIGENCE%'
 AND EXISTS (
```

```
   SELECT 1
   FROM contact_activity ca
   WHERE ca.accid = c.id
    AND ( LOWER(ca.name) LIKE '%rogers%')
      OR LOWER(ca.email) = 'eliottpamela@gmail.com')
    AND EXISTS (
      SELECT 1
      FROM participants p
      INNER JOIN `user` u ON p.userid = u.id
       WHERE p.actid = a.id
        AND LOWER(u.name) LIKE '%james%')
        AND LOWER(c.name) LIKE '%collins%'
```

The query described in Listing 3.9.2 is an implementation of the current scenario for N1QL.

Listing 3.9.2 An N1QL implementation for the Search criteria scenario

```
SELECT meta(a).id,
 a.title,
 a.startDate,
 a.account.name,
 a.contacts,
 a.participants
FROM crm a
WHERE a.type='activity'
 AND a.activityType='Appointment'
 AND SEARCH(a,
   {"conjuncts": [
      {"field":"title", "match": "artificial intelligence"},
      {"field":"contacts.name", "match":"rogers"},
      {"field":"contacts.email", "match":"eliottpamela@gmail.com"},
      {"field":"contacts.phone", "wildcard":"*6816*"},
      {"field":"participants.name", "match":"james"},
      {"field":"account.name", "match":"collins"},
      {"field":"startDate","start": "2016-08-29",
        "end":"2016-08-30", "inclusive_start": true, "inclusive_end": true}
       ]
   }, {"index":"all_acts"})
```

The query described in Listing 3.9.3 is an implementation of the current scenario for a MongoDB query.

Listing 3.9.3 A MongoDB query implementation for the Search criteria scenario

```
db.activity.aggregate([
  { $match: {
      $text: {
        $search: "activity Appointment \"artificial intelligence\" 6816
rogers eliottpamela@gmail.com james collins" },
        "type": "activity",
        "activityType": "Appointment",
        "title": { $regex: /artificial intelligence/ },
        "contacts.name": { $regex: /rogers/ },
        "contacts.email": { $regex: /eliottpamela@gmail.com/ },
```

```
        "contacts.phone": { $regex: /6816/ },
        "participants.name": { $regex: /james/ },
        "account.name": { $regex: /collins/ },
        "startDate": { $gte : '2016-08-29', $lte : '2016-08-30' }
  }},
  { $unwind: "$contacts" },
  { $project: {
        _id: 0,
        title: 1,
        startDate: 1,
        accountname: "$account.name",
        contactname: "$contacts.name",
        contacttitle: "$contacts.title",
        contactemail: "$contacts.email",
        participants: 1
  }}
])
```

Summary

The main point in this query is the search criteria. All queries languages offer the necessary functionality for implementing this query. The MySQL implementation is based on `LIKE`. N1QL has the Search Service which supports such a query. A MongoDB query uses `$search` and `$regex` for implementing this query.

In MySQL, the `LIKE` operator is used in a `WHERE` clause to search for a specified pattern in a column. This query uses `LIKE` each time it tries to search any values in any position. In MySQL, each criterion should be specified after the `AND` operator.

In spite of N1QL being an SQL-like language, it offers a solution other than `LIKE`, which is `SEARCH()`. This method has *conjuncts* if several criteria in search already exist. This means criteria can be just separated by commas.

A MongoDB query also has the `$search` function, where values have to be specified. However, a MongoDB query searches these values in each field. After a MongoDB query has collected the results from the search, the `$regex` function should be used to define a search pattern.

All the databases can get the desired result as each database offers different functionality for implementing the search request. MySQL uses `LIKE`, N1QL uses search with `conjuncts`, and a MongoDB query uses `$search` with `$regex`. The query on each of the languages is not complex. With N1QL, the query is more expressive compared to MySQL and a MongoDB query.

Table 3.9.1 Metrics for the Search criteria scenario

Criteria	MySQL	N1QL	MongoDB query
Simplicity	7	7	7
Readability	8	9	8
Expressiveness	9	9	8
Flexibility	8	10	9
Skills availability	4	6	5
A number of code lines	24	21	26
A number of client/server trips	1	1	1

4. Conclusion

Not all the three query languages can meet all the requirements of any given scenario. Only N1QL was able to accomplish all the scenarios. A MongoDB query and SQL can achieve eight out of nine scenarios and six out of nine scenarios, respectively. Each solution has its advantages and disadvantages that become more or less apparent depending on a specific criterion to meet.

N1QL demonstrates good results across all the evaluated scenarios and appears to be a good choice. Despite the nature of Couchbase Server, N1QL is quite similar to SQL. Furthermore, N1QL offers a number of extra features useful in implementing unconventional and complex scenarios. Though, N1QL provides a broad set of features, but it still requires extensive skills for handling this functionality.

SQL demonstrates rather good results across all the scenarios. SQL is quite simple from the developer's perspective, so its implementation across all the scenarios does not involve much complexity and has a good expressiveness. It should be noted that in some scenarios SQL lacks flexibility due to the nature of database.

A MongoDB query produced comparatively low results. The reason of such performance is the limitation of a MongoDB query. In most scenarios, a query implementation consists of several subqueries. This way of implementation leads to a big number of client/server trips and code lines, which has an impact across all the criteria. Despite limitations, a MongoDB query offers build hierarchy on the fly. This allows for simplifying a JSON model.

5. About the authors

Artsiom Yudovin is a Data Engineer at Altoros with a solid software development background. He is focused on maintaining, designing, customizing, upgrading, and implementing complex software architectures, including data-intensive and distributed systems. Artsiom dedicates much of his spare time to these activities and now he is one of the contributors to well-known open-source projects.

Uladzislau Kaminski is a Senior Software Engineer and Cloud-Native Development Consultant at Altoros. His primary skills are software architecture and system design. He took part in a numerous of projects connected with processing and distributing huge amounts of data arrays. Uladzislau has a durable background in building systems from scratch and adapting existing solutions, as well as designing, analyzing, and testing them.

Altoros is a 300+ people strong consultancy that helps Global 2000 organizations with a methodology, training, technology building blocks, and end-to-end solution development. The company turns cloud-native app development, customer analytics, blockchain, and AI into products with a sustainable competitive advantage. Assisting enterprises on their way to digital transformation, Altoros stands behind some of the world's largest Cloud Foundry and NoSQL deployments. For more, please visit www.altoros.com.

To download more research papers and articles:

- check out our resources page
- subscribe to the blog
- or follow @altoros for daily updates