CHAPTER 2

REVIEW OF LITERATURE

2.1 Iceberg Query Processing Techniques

The Iceberg query processing is first described by Fang et al. (1998). In this research, the author proposed different techniques to process iceberg query. The first technique is based on sampling concept. This method takes random samples from the input dataset. The size of the sample is based on the threshold value. It aggregates and extracts the records from each such a sample. Then it selects the sample which becomes a candidate for the final iceberg query result. The same procedure is repeated for all the samples. This method is straightforward to implement but occurs both false positive and false negative results. To overcome these errors, they proposed hybrid methods.

The second technique is based on coarse count or probability based count. It uses an array of counters and hash functions. A hash function is used to map target values. In this case, linear scanning of relation is done. Then for each tuple match with a target value, the counter is incremented. This procedure is repeated until the end of the dataset for all query attributes. After this, hashing is applied to generate bitmap through an array of counters. Rather than allocating a counter for each distinct value, it allocates a counter for a group of distinct values. It uses a hash function to retrieve the values into groups, and this is the objective of this method. These building blocks produce false positive, values that are considered as candidates for the final solution.

Sampling causes false negatives values whereas coarse count did not suffer from this problem. They develop hybrid algorithms like Defer count, multilevel and multistage algorithms to overcome these problem. These algorithms
have also overcome the limitations of the study proposed by Whang et al. (1990), which only uses probabilistic techniques to evaluate iceberg query and faces the problem of false negative value. Several optimisation methods are presented by Fang et al. (1998), that uses a combination of multiple hash functions and multiple scans over relation R. Use of multiple scans reduces the number of false positive significantly but takes more time. Thus, the method that requires less number of table scan is the overall best. Because, it is not only produced less false positives, but it was also quicker. In this way, this research evaluates iceberg query using small memory and less number of passes on the relation. The problem with defer count algorithm is a selection of sample and identifying the most frequent target. The difficulty with the multilevel algorithm is deciding the criteria to split the amount of main memory between two structures. The performance of above strategy depends upon sample selection and frequent target.

Bae and Lee (2000) have proposed iceberg query processing method. This research concentrates on evaluation of AVERAGE function in Iceberg query by using partitioning technique. They developed basic partitioning and postponed partitioning algorithm to compute an AVERAGE function of Iceberg query. The basic concept behind this algorithm is to partition database logically to find candidate set of tuples which are satisfying threshold condition of Iceberg query. These algorithms are worked in two phases. First, it partitions relation and select candidate and second computation of the average value of candidate set. Postpone partitioning tries to postpone partition of the database which affects number and size of the partition. Due to this, the chances of having data below the threshold in the partition is less in case of POP than basic partitioning (BAP) method. It has been proved that performance of these algorithms depends upon data order and memory size. The performance of this method is outstanding if the input relation is in sorted order.

Collective structure of Iceberg query processing has proposed by Leela et al. (2004). In this paper, researchers developed different Iceberg query processing
technique and suggest algorithm that can be implemented in query optimizer. It helps to select a more relevant algorithm for Iceberg query evaluation (IQE). Collective Iceberg Query Evaluation (CIQE) is work on sampling and hashing concepts. The performance of CIQE is measured using three algorithms such as Sort Merge Aggregate, Hybrid Hash Aggregate and ORACLE. In case of data set with the average target and high data skew, the performance of CIQE is better than SMA. HHA performance was poor when the number of targets was high, and it has many implementation difficulties. There was a considerable performance gap between the online algorithms and ORACLE, indicating a scope for designing better iceberg query processing algorithms.

Beyer and Ramakrishnan (1999), Agrawal et al. (1996) and Han et al. (2001) described the concept of iceberg cube computation. These methods calculate and identify the cell of a data cube which satisfying threshold condition. Based on the aggregation attribute these methods focused on applying a proper sequence of operations to be performed. This help to make the best use of the sharing of calculation. Iceberg cube computation totally depends upon the performance of iceberg query.

To maximize the computation speed of iceberg cube by improving the performance of iceberg query, Ferro et al. (2009) have designed a two-level bitmap index which can be used for processing iceberg query. By considering the applicability of bitmap indexing, this research makes use of bitmap indexing for iceberg query evaluation. This is the first research which makes use of bitmap indexing for iceberg query evaluation. However, this strategy suffers from empty bitwise AND operation problem. The algorithms proposed by this research can be used to create iceberg cube efficiently.

Dhande and Bamonte (2014) describe a technique for evaluating query by finding subquery. The basic intention of this research is to optimize the query
evaluation process. However, the limitation of this approach is it is not suitable for the application where data retrieval rate is more.

Above mentioned approaches suggested are come under the group of the tuple scan based method. None of the above research considered properties of iceberg query for its evaluation. These algorithms focus on reducing the number of tuple scan, but none of them makes use of Iceberg query property. These strategies are work on the concepts like:

1. **Applying aggregation concept:** It first aggregates all tuples on given query attribute and then it performs HAVING clause to retrieve iceberg query result. Along with the HAVING clause threshold value check is performed to eliminate the tuples which do not contribute to the final query result.

2. **Multi-pass aggregation concept:** The multi pass aggregation concept is useful when size of the database is more and it will not fit in main memory. In this case the data set is divided into some subsections. On each subsection, the aggregation concept is applied.

3. **Tuple scan based:** All above approaches are tuple scan based. Each of them needed full table scan which is very difficult. In case of scanning of full table, first it must be loaded in main memory which is not feasible in data warehouse. The backend database of data warehouse is very huge. Due to this, tuple scan algorithm takes too much time to execute iceberg query. Above algorithms does not consider properties of Iceberg query for its evaluation. These algorithms focus on reducing the number of tuple scan, but no one of them makes use of iceberg query property for its evaluation.

However, Ferro et al. (2009) try to make use of bitmap indexing for Iceberg query evaluation. This research partition the main dataset horizontally concerning leftmost ordered dimensions. It generates the two-level bitmap vector
on the dataset. Then by performing bitwise operations on bitmap vector, the aggregate function is evaluated. This research explains detail algorithm for COUNT aggregate function and proposes an extension of the same for SUM, MAX and MIN function. But this strategy undergoes from empty bitwise AND operation problem.

This problem is minimized by Bin et al. (2012) by using dynamic pruning and alignment of vector approaches. Both these approaches make use of the antimonotone property of iceberg query and support only the antimonotone aggregate functions. Dynamic pruning algorithm is work in this sequence. First up all, it performs bitwise AND operation on given query attributes. Then to eliminate current fruitless vectors and to generate new vector it uses XOR operation. This procedure is repeated till all the vectors complete. Simultaneously based on the result of AND and XOR operation it generates final Iceberg result. In this research, bitwise AND is performed in the order in which they are present in bitmap index. Here, they only consider the order of vector without its value. This is the major cause which increases empty bitwise AND operation. Compare to the algorithm proposed by Ferro et al. (2009), it minimizes fruitless AND operation problem by implanting index pruning strategy. But it does not have control over fruitless XOR and OR operation. Also, queue maintenance cost is the major problem of this research.

The problem occurred in two-level bitmap index introduced by Ferro et al. (2009) and dynamic pruning strategy of this research are overcome by using methods developed by Bin et al. (2012). They come up with the alignment of vector concept. It starts with the creation of bitmap index on query attribute. Then it searches for alignment of 1's bit position. As per the sequence of 1's bit position it applies the dynamic pruning concept on aligned attributes. After each set of bitwise AND operations threshold checks condition is executed. Based on its result further XOR operation is performed. According to the result of XOR again the position of 1’s bit of vector gets changed. As per the new sequence, all above steps are executed till the end of vectors. In this way this algorithm minimizes the problem
of bitwise AND but fruitless XOR problem arises in this case. At the same time, queue maintenance cost is increased in this case which affects the performance of iceberg query.

Rao and Shankar (2012) come up with technique to handle empty bitwise XOR operation problem raised in Bin et al. (2012). The intention of this study is to minimize the fruitless bitwise XOR operation by postponing its execution. They have added the phase of counting number of bits of bitmap vector of query attribute. The XOR operation is postponed even though count satisfy threshold condition but positions are not in sequence. Similarly, if the difference number of 1’s bit is less than the threshold value then the current vector is prune from bitmap index. Such vectors will not be considered again for this query. In this way, this research postpones the XOR operation, and it helps to reduce fruitless XOR operation. But this study does not focus on minimizing fruitless bitwise AND operation.

Shankar and Guru (2012) described the method to minimise fruitless operation using a random selection of bitmap vector for processing and to remove the zero bit from the sequence which is not aligned with the adjacent sequence. This technique is not efficient if data is skew and random selection of attribute is not effective every time.

Laxmaiah et al. (2013) described a method to process aggregate queries effectively by applying the concept of the priority queue. By assigning priority to vector, they are minimising fruitless operation. But this study did not have control on queue push in and popped out the operation. It will be time-consuming if input data set increases.

Praveen and Vuppu (2013) presented array based indexing strategy for execution of iceberg query. This study initially created bitmap index and then on that it applies the array indexing technique. They are not performing bitwise AND
operation. This study makes use of array index and comparing index value with adjacent bitmap vector. This process is very complicated if data size goes on increasing because for every bit it performs a check for 1's bit. As per the presence of 1's bit, it creates the array index of position. Finally, it matches the index of attribute vector and declares the query result. This approach is only suitable for COUNT aggregate function.

Vuppu and Guru (2013) illustrated the different approaches for iceberg query processing. The base of these approaches is position of 1’s bit in bitmap index, the density of occurrence of 1’s bits in vector and control the queue push and pop operations. They tried to minimise fruitless operations as well as queue maintenance cost. The concept which this study described is useful for COUNT aggregate function.

Zhian et al. (2013) have discussed the effective approach to evaluate iceberg cells of s-cuboids. The result of Iceberg pattern based aggregate query is sequence cuboid which is a multidimensional array of cells. Every cell of s-cuboid is associated with the pattern template of the query. The value of each cell is calculated by aggregating the data set of the sequence as per query pattern. The cuboid computation completely depends upon the iceberg query evaluation method used.

The algorithms based on a set theory for iceberg query execution is proposed by Chandra and Sammulal (2014). This algorithm generates the set of each attribute vector as per the position of 1’s bit. The global matching strategy is applied on all sets to find the sequence of attributes to be considered for the operation. Based on this sequence union and intersection operation is performed on set. The result of all subsets is combined with threshold condition check. It will be included in final iceberg query result if it satisfies threshold condition. The queue is required to maintain each set. This strategy does not have control on queue
operation, so it affects on the performance of the query. This strategy considers only COUNT aggregate function evaluation.

Xiaochun et al.(2015) described a strategy that supports fast evaluation of queries with aggregate function and range constraint. It divides the data into different parts using balance partition approach. Estimation of each partition is prepared. It is a metadata of each partition. The analysis of all partitions has to be performed when range aggregate query arrive. This analysis is done by considering metadata of each subsection. Range constraint help to get the result by joining outputs of all partitions. But here overhead of Merge-Join operation occurs which degrades the performance of the aggregate query. Creating, storing and retrieving metadata is additional overhead which degrades the performance of this strategy.

All above research of evaluating iceberg query uses queue concept for all vectors but all of them faces the problem of futile queue push in and pop out. They have only suggested the application of queue to reduce fruitless operations but none of them thought about maintenance of queue. This is the major drawback of above strategies. Due to this problem of minimising fruitless bitwise AND and XOR operation is not solved by them. All these research only described their strategies for COUNT aggregate function. None of them describes methodology for an aggregate function which does not have antimonotone property.

The limitations of all above research can be minimized using bitmap preprocessing strategy for Iceberg query evaluation. The basic concept of this research is preprocessing of the generated bitmap. Based on the sequence of query attributes, bitmap vector is generated, and it is preprocessed as per the requirement of the query. Due to this the vectors which are having the highest probability of to be a part of final answer are evaluated first. The comparison of the result with next highest vector is made so the chances of pruning the vector in advance will be more. This process minimizes fruitless bitwise AND operation, XOR operations and futile queue operation problem. None of the previous research work on other
aggregate functions like SUM, MIN, MAX and AVERAGE. This research provides complete framework for iceberg query with COUNT, SUM, MIN, MAX and AVERAGE aggregate function.

Zhian et al. (2017) described pattern based aggregation on sequence data. The result of pattern based aggregation is a sequence of cuboids which is represented in multidimensional array of cells. This research uses Iceberg query to represent each cell of s-cuboid. The value of each cell is computed by evaluating respective Iceberg query. Generally sequence data is very huge and may contain more number of attributes. But for analysis the cells with large aggregates or as per threshold value are required to find pattern. In this way this research uses Iceberg query to identify pattern of sequence data.

Brett et al. (2017) proposed the optimizing technique for Iceberg queries with complex join. Iceberg queries itself are very complex and expensive to evaluate on huge data set. Multi join operations increase the complexity of Iceberg query. This research provides the solution to evaluate Iceberg query on multi join operations. It uses the concept of a-priori, pruning and memorization.

Yoonjae et al. (2017) introduced mechanism for efficient processing of skyline query. Similar to Iceberg query, skyline query is also used for multi criteria decision support applications. Skyline queries are worked on multidimensional spaces and streaming environment. The numbers of objects returns by skyline queries are more. The patterns of object generated by skyline queries are identified using Iceberg query. Input to the skyline query is large data sequence on distributed environment therefore this research uses MapReduce concept to evaluate the skyline query.

Hsueh et al. (2017) described an efficient indexing method for evaluation of skyline query. This technique is based on use of cache for storing query result as well as user preferences. Along with query result, its preferences become the index
to search result of respective query from cache. In this way query processor can rapidly retrieve the result for a new query only from the result sets of cached queries with compatible user preferences. This indexing mechanism improve the performance of skyline query in terms of time and data storage required.

Ekateriani and Minos (2015) described a technique of query analytics over probabilistic databases with unmerged duplicates. This research introduced entity join operator that allows expressing complex aggregation and iceberg queries over join between tables with unmerged duplicates and other database tables. In this way they suggest novel indexing structure for efficient access to the entity resolution information and novel technique for the efficient evaluation of complex probabilistic queries that retrieve analytical and summarized information.

Rui et al. (2017) suggested the self adaptive partition framework for improving performance of top-k queries over streaming data. Generally, top-k queries are evaluated on each window from partition. Partition contain number of windows as per the data size. Evaluation of top-k query is done by processing iceberg query on each window. The result of Iceberg query is used to declare which window belongs to top-k query. In this way evaluation of top-k query is very complicated task. The evaluation of top-k query is simplified using the concept of self adaptive partition.

2.2 Role of Aggregation Function in Iceberg query

Aggregation function can summarize the data. Summarization of data with some constraint on the attribute is the intention of iceberg query. So aggregation functions are an integral part of iceberg query.

As stated in research article by Gray et al. (1997) there are three different type of aggregate functions. Consider a set of Tuples S_T and set \{S_j | j=1...n\} to be any complete set of disjoint subset of S_T. Distributive aggregate function \( \text{AggF} \)
is distributive if there is a function $G$ such that $\text{AggF} (T) = G (\{F (S_j)| j = 1 \ldots n\})$. MAX, MIN and SUM are distributive with $G = \text{AggF}$. The count is distributive with $G = \text{SUM}$.

An aggregate function $\text{Fun}$ is algebraic, if there is an $N$-tuple valued function $G$ and a function $H$ such that $\text{Fun} (T) = H (\{G (S_j)| j = 1 \ldots n\})$. Standard deviation, Average, the centre of mass, MIN_N, and MAX_N all are algebraic aggregate functions. In case of Average aggregate the function $S_C$ records the sum and count. The $F$ function adds these sums and count components and then divide to produce the overall average value of the set. A similar method is required to find the $M$ greatest value, the centre weight of group objects from the set and other algebraic functions. An aggregate function $F$ is holistic if there is no constant limit on the size of the storage needed to represent a sub-aggregate value. That is, there is no constant $N$, such that an $N$-tuple characterises the computation $F$.

The efficient process of evaluating OLAP queries with aggregate function and join operation is explained by Kim et al. (2002). This method is suggested for data warehouse system where a large volume of input dataset is required. In such an environment executing query independently is a very complicated task. They make use of properties of aggregate function to improve the performance of data warehouse query. This strategy is useful for executing OLAP queries on star schema.

Ryan et al. (2007) demonstrated the importance of aggregation function in the distributive environment. They evaluate continuous aggregative function by distributing it among different subsets. Based on result and property of aggregate function the final result of aggregation is generated. Aggregation is essential in distributed query processing also.

Cesar et al. (2008) contributed execution methods of star join queries. These queries are executed on data warehouse in Microsoft SQL server. These star
join queries consist of aggregation function and join operation. They use features of aggregation function for implementing bitmap filter technique. This research contribution is added in SQL Server 2008.

Paulo et al. (2011) described that executing aggregate queries are very complex and tedious work. The tremendous increase in data size has brought a big challenge to evaluate the aggregate function on it. Different characteristics of aggregation classes are described in detail. They studied different distributed data aggregation methods and found the limitation of it.

Issam et al. (2014) presented a research contribution to enhancing the performance of real-time data warehouse system. They implemented the algorithms for management of materialised views dynamically. This process is performing using queries with aggregate function and multiple joins. Based on the feature of aggregate function of query the materialize views get updated dynamically. Here aggregate function results are the latest results which are easily reflected in the materialized view. In this way by making the use of aggregate function from query this research update their materialized views.

Performance analysis of row and column storage data warehouse is contributed by Archana and Suresh (2015). Row based data storage is useful for OLTP and column based storage is useful for OLAP. Update and delete operation is complex in case of column store architecture. The aggregate function required same data related to particular attribute which is easily available with column store architecture. An aggregate function is essential in data warehouse query and data of data warehouse is not frequently changed therefore column based storage is suitable for data warehouse implementation.

Raam et al. (2017) described fast range aggregate queries for big data analysis. The way of executing range aggregate is matched with Iceberg query because both these queries consist of aggregation function. In range query the range
of aggregation is specified whereas in Iceberg query the threshold value check is specified. Hence both these queries are very much useful for big data analysis. The researchers have applied the concept of partitioning and then evaluating the aggregate function on each partition. The final result is obtained by grouping the values from all the partitions.

Efficient computation of all these aggregate functions is very important. Execution cost of these aggregate functions regarding time is much higher than that of the other basic relational operations like SELECT and PROJECT. Thus the performance of iceberg query directly depends upon the aggregate function.

2.3 Bitmap Indexing a Suitable Technique for Iceberg Query Evaluation

A bitmap index is a suitable approach for querying huge, multidimensional and historic datasets. It has drastically increased query performance and reduces data accessing time on high as well as low cardinality attributes sets. Bitmap index help to process iceberg query efficiently as it possesses special characteristic. It avoids vast disk access for complete tuple sets. It only accesses bitmap vector of attributes specified in the group by clause. Bitmap operates on bits rather actual tuple values. Bitwise operations are very fast to execute as they directly supported by hardware computer system. The bitmap can adopt the antimonotone property of Iceberg query as well as aggregate function very easily. This helps for index pruning during Iceberg query evaluation. A bitmap index on given relation can be created by combining the bitmap vector of all attributes. Knuth (1973) described the applicability of bitmap index to generate inverted files.

Summary table approach has been introduced by O’Neil and Quass (1997). It is based on creation and store of aggregated and summarized data. This is suitable only for prediction based queries. By considering predication attribute, the
summarized tables are computed, and results are generated. To evaluate the query, this strategy does not use data from the data warehouse. Instead, it uses summary tables. Based on prediction attribute generating summary table is a complicated and time-consuming task.

Development of the bitmap index generation and compression introduced by Wu et al. (2006), Deliege and Pedersen (2010) and Antoshenkov (1995). These algorithms show that bitmap index in compressed format require less memory compare to the original relation. Along with compression they also introduced encoding strategy to enhance the applicability of bitmap index in different applications. The performances of above techniques are not better for high cardinality attribute.

Stockinger et al. (2004) described a partition based approach to solve the problem of high cardinality. The size of a bitmap index is small in case of low cardinality attributes. But for high cardinality attribute, it increases tremendously. Operating on such a huge bitmap is very complex. In such a case bitmap index is created with bins. Attribute values are divided into different bins and bitmap index is generated bin-wise. This help to reduce the storage space as it uses binning approach.

Wu et al. (2004) and Stockinger et al. (2009) described that bitmap indices are suitable for different type of high cardinality attributes like numeric, categorical and text. They also showed that performance of bitmap index is better for different types of queries like aggregate, range, equality, membership and keyword-based query. The value mentioned in predicate condition is read into memory to generate the result of equality and keyword-based queries. Whereas in case of membership, range, equality and aggregate queries, if more than one bitmap vector is present then bitwise OR operation is performed \((n-1)\) time on complete bitmap vector, where \(n\) is the size of bitmap vector.
Bitmap encoding method has been studied by Yuhao et al. (2014). This method generates symmetry encoding based on occurring of 0’s and 1’s in bitmap index. But if the occurrence of number 0’s is more, the performance of indexing degrades. This algorithm is developed for recording of continuous data.

Zhen et al. (2015) described the analysis of different bitmap index compression algorithms for big data. The different features of bitmap index are it can easily compressed, increase speed of logical bitwise operations and reduces storage requirement. Therefore, it is suitable for the applications like geographical information system, graph databases, image retrieval, Internet of things, etc. All these applications work on big as well as continuous data where bitmap indexing is suitable.

Lemire et al. (2010) introduced sorting technique to improve word aligned bitmap indexes. They used the technique based on run length encoding such as word aligned hybrid for bitmap indexes. The compressed bitmap reduces input output cost and minimizes CPU usage. A simple lexicographical sort can divide the index size and make the individual indexes several time faster. It further increase the efficiency of sorting by permuting the columns. They found that 64 bit indexes are slightly faster than 32 bit indexes.

Witold and Robert (2011) has proposed graphical processing unit based position list word aligned hybrid algorithm for bitmap compression. A drawback of bitmap index is that its size increases when the domain of an indexed attribute increases. Hence the bitmap index compression algorithms are required. In this case execution of compression as well as decompression algorithms is required to maintain bitmap index. This study proposed parallel algorithm for bitmap index compression and decompression which reduces time of bitmap processing.
A new compression scheme for bitmap indexes has described by Zheng et al. (2017). They proposed compression dirty snippet method which work on the concept of word aligned hybrid. The basic idea is to trade some payload for flexibility so that the probability of space reduction is raise, which achieves better compression.

Yinjun et al. (2016) introduced a new bitmap index coding algorithm for big data. They developed combining binary and ternary encoding, a new bitmap coding algorithm. It is derived from word aligned hybrid and compressed adaptive index. It combine both these mechanisms and results in more compact bitmap indexes.

In today's era, due to the involvement of advanced and scientific computing in all fields of engineering and sciences, the processing of application generates a huge number of parameters related to that application. Ying et al. (2013), Imho et al. (2008), Dellaquila et al. (2006), Inmon (2005), Oracle White paper (2011), Oracle White paper (2014) and Oracle White Paper (2015) described that data warehouse collects huge information related to application from various external data sources. It is subject oriented and historic collection of databases. A data warehouse is not updated frequently. Decision support system of organization is work on platform created by the data warehouse. They analyzed the data and based on analysis take the decision related to the organization. A data warehouse is huge and complex, effectively retrieving information from such a huge database is the challenge in front of the researchers. The nature of the query to be fired on data warehouse is aggregation function followed by group by clause with some threshold constraint. Such a type of queries is called as iceberg query. A data warehouse is huge, directly executing a query on such a huge dataset is time-consuming and is not effective. Efficient indexing techniques are required to search data from the huge database.
Marcus and Lenz (1999), Rishi et al. (2006), Doron et al. (2006) and Morteza et al. (2008) described that indexing is very important in the database system to increase the performance of database query. The index is assigned to columns when it is created. Attribute index help to search particular tuple from the relation. The indexing techniques used in database systems are B tree, Hash, R Tree and bitmap. Out of these techniques hash, B Tree and R Tree are applicable for online transaction system(OLTP), and bitmap indexing is suitable for Online Analytical Processing(OLAP). The bitmap is created only on the query attributes. It has to be created only one time. Next time if new query required the same attribute then it will use previous bitmap only. Due to this, it saves time and also avoids full table scan. Generating the bitmap index of the attribute will not affect the performance of query because generated bitmap by the database system is in compressed mode. A data warehouse is having properties that it works on data set which is in read mode only, updates are not frequent and it is huge. Due to these properties of data warehouse ,bitmap indexing is the suitable approach for it.

Niwan and Sirirut (2006) and Weahason et al. (2009) introduced the strategy based on bitmap indexing which is suitable for membership and equality query. They created the dual bitmap index by using interval attributes. To improve the efficiency of scattering bitmap they use data clustering methods. Interval value and cluster value help to index membership and equality query only. These techniques are suitable only for membership and equality query.

Improved bitmap indexing method for data warehouse system is developed by Navneet et al. (2006).The objective of this study was to speed up data warehouse query. The attribute of relation considered for bitmap index creation are sorted, and then bitmap index is created. A generated bitmap index is then compressed. They noted significant performance improvement in case of equality and range queries.
Bin and Peng (2011) developed indexing strategy for data processing in the cloud. They have created two bitmap index on two level. One is on local node site and another at global node site. It is like indexing for the index because the data flow on the cloud is distributed. Each site node has the fast indexing technique to search data on the cloud.

Peng et al. (2013), Abdelouarit et al. (2013) and Naveen and Rai (2013) described the suitability of bitmap indexing for data warehouse, large data store and data mining. Along with bitmap compression, they adopted the partial indexing scheme. In this case, dynamically the bitmap gets created if updates are there in the main database. These concepts support high cardinality vectors as well as range and join queries.

Parth (2015) introduced hierarchical bitmap indexing method. This concept is applied to column store architecture. Column store architecture is efficiently handled bitmap indices. Column attributes are in hierarchical nature, so this indexing is suitable for column store architecture. This method is well suitable for range queries.

Dani and Trivadis (2011), Oracle (2015), Abdelouarit et al. (2014), Oracle (2014), Oracle (2011) and ONeil (1987) explained that bitmap indexing is the suitable approach for large databases, data warehouse and data mining applications where the database is not frequently changed. Oracle's innovation in bitmap indexing is widely used for large database applications. Oracle implant bitmap index and compressed bitmap index in its architecture. They described the suitability of bitmap indexing for processing aggregate, complex and multidimensional queries. Different commercial products of database system like Oracle, IBM, and Sybase introduces bitmap index in their architecture.
2.4 Role of Cache in Query Processing System

The idea of loading cache with query result has been introduced by Haas et al. (1999). They integrated caching in their optimizer architecture. Query results are loaded into the cache as it is. They have not used any retrieval mechanism, so the size of cache creates the problem. This strategy is well suitable for small result and data set.

Selvaraj et al. (2006) proposed query optimization with caching for distributed databases. They apply this technique to heterogeneous and distributed databases. The query contained in the cache is in XML format. They used XML parser to accept SQL statement. The result of XML parser is matched with cache XML format, and the query result is obtained. In this way, it improves the query performance on distributed databases.

The performance of cache can be improved using an estimation of query result size for proxy cache has introduced by, Tanu et al. (2006). Before inserting query result into the cache, it uses query estimation to find the size and frequency of query. Based on size and frequency it decides the location of the query result in cache. Thus maximum utilization of cache store is possible.

Optimal cache document replacement strategy has developed by Toly (2006). This strategy tries to keep minimum load on a cache by using novel document replacement algorithm. Due to this, the utilization of cache is increased and searching and retrieval time gets reduced.

Barla et al. (2012) described the applicability of cache mechanism for query processing in the search engine. The intention of the researcher is to reduce the query processing time on a search engine. In this study, they used the indexing technique on the cached result. Indexing is required in this case because it worked
on large data set. Based on the index result it searches query result quickly from the previous results stored in the cache.

Mantu et al. (2012) proposed a method of query optimization in distributed database using cache technique. They proposed the technique by considering four issues like the capacity of the server, a load of the server, distance and length of the queue. They found better hit ratio even for first time execution of the query. Performance of query is increased in this case.

Kumar and Vaideswaran (2012) described a semantically based cache method to optimize database query. This study concentrated on effective utilization of cache and efficiently retrieved contents from the cache. They used the concept of cache in the proxy server as well as in web server. The semantic analysis rules are defined and stored on the server. As per the query evaluation strategy, the semantic analysis of query is mapped to the cache content and fetches the query result from cache. In this way, they optimize the query evaluation process.

Caching support for skyline query has described by Hsueh and Tristan (2014). It is challenging to process skyline query for online applications with low response time. Individual skyline query performed on data sets with partially ordered domains. This research suggest to cache the result of skyline query and retrieve it from cache if same query occurs. They introduced similarity measures used to find similarity between the user preference of a new query and a cached query.

Dunlu et al. (2013) proposed caching mechanism for aggregate query evaluation on Map Reduced platform. This strategy searches the aggregate result references in the cache of clusters. It has combine result from the different cache if it found the answer to query then immediately it responds. In this way, it avoids processing of query again and again. It improves the performance of the query regarding time.
In this way, different researchers have used cache for query processing strategy. Their result shows that performance of their strategy is improved by using the cache.

2.5 Existing Method to Evaluate Iceberg Query

This subsection demonstrates the existing method to evaluate Iceberg query. Traditional methods to evaluate Iceberg query using bitmap index is described by Bin et al. (2012). They used the bitmap indexing concept because iceberg query is evaluated on a large dataset, its structure is very complex and it needs repeated access to the database during its evaluation. Traditional method treat Iceberg query as normal query. The problems faced by this method is fruitless bitwise AND, OR and XOR operations. It also goes through the futile queue pushing problem. Due to these problems, the overall performance of iceberg query get degrades regarding time, iterations and database access required to evaluate iceberg query. Traditional iceberg query processing techniques are time-consuming as they all work on bitmap index directly. They are not considering the properties of iceberg query while evaluating it. They have to perform bitwise operations repeatedly, and most of the time the results are fruitless. All these factors directly affect the performance of the iceberg query. Following example demonstrate the detailed workflow of traditional iceberg query processing technique. Assume the iceberg query with COUNT, SUM, AVG, MIN and MAX function as shown in Table 2.1.

Table 2.1 Example Iceberg query

<table>
<thead>
<tr>
<th>Query No.</th>
<th>Aggregate Function</th>
<th>Query</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>COUNT</td>
<td>Select Month, Category, Count(<em>) from Monthly_Exp_25 group by Month, Category having Count(</em>)&gt;=4;</td>
</tr>
<tr>
<td>2.</td>
<td>SUM</td>
<td>Select Month, Category, SUM(Amount) from Monthly_Exp_25 group by Month, Category having SUM(Amount)&gt;1000;</td>
</tr>
<tr>
<td>3.</td>
<td>AVG</td>
<td>Select Month, Category, AVG(Amount) from Monthly_Exp_25 group by Month, Category having AVG(Amount)&gt;600;</td>
</tr>
<tr>
<td>4.</td>
<td>MIN</td>
<td>Select Month, Category, MIN (Amount) from Monthly_Exp_25 group by Month, Category having MIN (Amount)&gt;200;</td>
</tr>
<tr>
<td>5.</td>
<td>MAX</td>
<td>Select Month, Category, MAX (Amount) from Monthly_Exp_25 group by Month, Category having MAX (Amount)&gt;500;</td>
</tr>
</tbody>
</table>
These queries can process on different size of relations. Here the demonstration is shown on input relation Monthly_Exp_25 which is as shown in Table 1.1. This table is created in Oracle and consist of 25 tuples.

According to the strategy of iceberg query evaluation first step is a generation of bitmap index on the query attributes. Attributes required to generate bitmap index are Month, Category and Amount as indicated in Table 2.1. These are main attributes which are involved in all iceberg queries shown in Table 2.1. The subset of the query attributes are required to generate bitmap index which is represented in Table 2.2.

<table>
<thead>
<tr>
<th>Query Attribute</th>
<th>Subset of attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>Month</td>
<td>{ June, July, August}</td>
</tr>
<tr>
<td>Category</td>
<td>{ Milk, Fruit, Vegetable}</td>
</tr>
<tr>
<td>Amount</td>
<td>[ 200,800,500,600,700,100,400,300]</td>
</tr>
</tbody>
</table>

The bitmap index generated on these attributes is as depicted in Table 1.2. The evaluation of all Iceberg queries specified in Table 2.1, is done on the generated bitmap index which is shown in Table 1.2.

2.5.1 **Query with COUNT Aggregate Function**

**Query 1**: Select Month, Category, Count(*) from Monthly_Exp_25 group by Month, Category having Count(*)>=4;

The evaluation of Iceberg query with the traditional approach is as follow. In case of COUNT aggregate function, it will perform bitwise AND operation between all the subset of attributes. As per query1, it has to find out the count of the same combination of Month and Category occurs in given database. Query result will be all combinations of Month and Category whose count is
greater than or equal to 4. As shown in bitmap index the first subset is June. This strategy will perform bitwise AND operations in the following sequence which is as indicated in Table 2.3.

**Table 2.3 Bitwise AND operations for query1**

<table>
<thead>
<tr>
<th></th>
<th>June</th>
<th>Milk</th>
<th>Result(R1)</th>
<th>June</th>
<th>Fruit</th>
<th>Result(R2)</th>
<th>June</th>
<th>Vegetable</th>
<th>Result(R3)</th>
<th>July</th>
<th>Milk</th>
<th>Result(R4)</th>
<th>July</th>
<th>Fruit</th>
<th>Result(R5)</th>
<th>July</th>
<th>Vegetable</th>
<th>Result(R6)</th>
<th>August</th>
<th>Milk</th>
<th>Result(R7)</th>
<th>August</th>
<th>Fruit</th>
<th>Result(R8)</th>
<th>August</th>
<th>Vegetable</th>
<th>Result(R9)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 2.3 indicates the bitwise AND between all the combinations of bitmap index. All the combinations are represented by result R1 to R9. The result R1 contains total number of 1's bit 5 times which satisfy the threshold condition specified in query 1. Therefore, combination (June AND Milk) is in the final Iceberg query result set. The Result R2, R3 and R4 contains 1's less than the threshold value hence these combinations cannot be a part of the final result of iceberg query.

In result R5 as indicated in Table 2.3, a total number of 1's are 5 which satisfy the threshold condition specify in query 1. Therefore, combination (July AND Fruit) is in the iceberg query result set. Result R6, contain 1's less than the threshold value, i.e. >= 4. Therefore (July AND Vegetable) combination cannot be a
part of the final result of iceberg query. Result R7 contain a total number of 1’s bit 4, which satisfy the threshold condition specified in query 1. Therefore, combination (August AND Milk) is in the iceberg query result set. Result R8 contains 1's zero times. Therefore (August AND Fruit) combination cannot be a part of the final result of iceberg query. Result R9 contain total number of 1's bit 4 times that satisfy the threshold condition specified in query 1, therefore combination (August AND Vegetable) is in the iceberg query result set. In this way, the iceberg query 1 is evaluated.

2.5.2 Query with SUM Aggregate Function

Query 2: Select Month, Category, SUM(Amount) from Monthly_Exp_25 group by Month, Category having SUM(Amount)>1000; The starting attributes as per the sequence of bitmap are June and Milk. To generate Query 2 result, summation of similar Month and Category combination has to be performed. The generated summation result must go to threshold condition check. The result is compare with threshold condition, if it satisfy threshold condition then it will be included in final query 2 result.

Here initially same steps of COUNT operation has to follow .Then bitwise AND that result with the Amount sub set i.e. 200,800,500,600,700,100,400,300. Compare to Query 1 evaluation, here one more iterative step is required because the aggregate function SUM has a different requirement. The computations performed for query 2 is as shown in Table 2.4.

Table 2.4 represents the bitwise AND for R1,R2,R4,R5,R6,R7 and R9.Result R3 and R8 contain all 0's , they cannot contribute to generate summation.
Table 2.4 Bitwise operation for query 2

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>200</th>
<th>S_R1_200</th>
<th>R2</th>
<th>200</th>
<th>S_R2_200</th>
<th>R4</th>
<th>200</th>
<th>S_R4_200</th>
<th>R5</th>
<th>200</th>
<th>S_R5_200</th>
<th>R6</th>
<th>200</th>
<th>S_R6_200</th>
<th>R7</th>
<th>200</th>
<th>S_R7_200</th>
<th>R9</th>
<th>200</th>
<th>S_R9_200</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

A.R1: June AND Milk

| Result(R1) | 1 0 0 0 1 0 0 0 0 0 1 1 0 0 0 0 1 0 0 0 0 0 0 |

To find the SUM of (June AND Milk) combination, bitwise AND between R1 and all the amount subset has performed. These bitwise AND operations are: (R1 AND 200),(R1 AND 800), (R1 AND 500),(R1 AND 600),(R1 AND 700),(R1 AND 100),(R1 AND 400),(R1 AND 300).

The results S_R1_200 contain two times 1 bit. Therefore, S_R1_200=200+200=400. Similar computation is performed for all the AND operations to find their sum value.

Thus the SUM(Amount) for (June AND Milk) combination is = S_R1_200+ S_R1_800+S_R1_500+S_R1_600+S_R1_700+S_R1_100+S_R1_400+S_R1_300 =400+000+500+000+000+100+000+300=1300

The SUM(Amount) for (June AND Milk) is 1300 which satisfy threshold condition given in query 2. Therefore (June AND Milk) combination is the part of final iceberg query 2 result.
**B. R2: June AND Fruit**

| Result(R2) | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Similar to the calculations done for (June AND Milk), the calculations for (June AND Fruit) combination has performed.

The bitwise AND operation performed are : (R2 AND 200), (R2 AND 800), (R2 AND 500), (R2 AND 600), (R2 AND 700), (R2 AND 100), (R2 AND 400), (R2 AND 300). The bitwise AND operation between (R2 AND 200) is as shown in Table 2.4. 

S_R2_200 contain only one time 1’s. Hence, S_R2_200 = 200. Similar operation is performed and sum value of individual combination is computed.

Thus the SUM(Amount) for (June AND Fruit) combination is = S_R2_200 + S_R2_800 + S_R2_500 + S_R2_600 + S_R2_700 + S_R2_100 + S_R2_400 + S_R2_300  

= 200+000+000+000+700+000+400+000  

= 1300

The SUM(Amount) for (June AND Fruit) is 1300 which satisfy threshold condition given in query 2. Therefore (June AND Fruit) combination is the part of final iceberg query 2 result.

**C. R3: June AND Vegetable**

| Result(R3) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

The combination (June AND Vegetable) contain all zero means this combination does not occur anytime in any transaction. Therefore there is no need to evaluate query for this combination.

**D. R4: July AND Milk**

The SUM(Amount) for (July AND Milk) is calculated by performing bitwise AND between R4 and Amount subset. The bitwise AND operation performed are : (R4 AND 200), (R4 AND 800), (R4 AND 500), (R4 AND 600), (R4 AND 700), (R4 AND 100), (R4 AND 400), (R4 AND 300).
The bitwise AND between (R4 AND 200) is performed as shown in Table 2.4. S_R4_200 contain all zero hence S_R4_200 = 000. Similarly remaining computations are performed and Sum value of R4 is calculated.

**NOTE:** Here it is noted that there is no need to repeat the bitwise operation between R4 and remaining subset of the amount because number of 1's present in R4 is only 3. Upto 6th step all three combinations are considered for summation. This parameter is not considered in the traditional approach they are repeating the operation till the end of attribute vector. Such a type of situations increases overheads in the traditional method and generate more fruitless operations.

Thus the SUM(Amount) for (July AND Milk) combination is = S_R4_200 + S_R4_800 + S_R4_500 + S_R4_600 + S_R4_700 + S_R4_100 + S_R4_400 + S_R4_300 = 000 + 800 + 500 + 000 + 000 + 100 + 000 + 000 = 1400

The SUM(Amount) for (July AND Milk) is 1400 which satisfy threshold condition given in query 2. Therefore (July AND Milk) combination is the part of final iceberg query 2 result.

**E.R5: (July AND Fruit)**

The SUM(Amount) for (July AND Fruit) is computed by performing bitwise AND between R5 and all the Amount subset. The bitwise AND operation is performed between (R5 AND 200), which is as shown in Table 2.4. S_R5_200 contain all zero means no combination of (July AND Fruit) has amount 200. Therefore S_R5_200 = 000. Similarly all remaining operations are performed.

**NOTE:** Here it is noted that R5 contains five 1’s, till the combination (R5 AND 700) summation of all five combinations is computed therefore further operations can be skipped. In this case threshold condition get satisfied after performing (R5 AND 500). Hence further operations can be skipped. But intermediate result check
concept is not considered in the traditional algorithm, and they repeat fruitless operations till the end of the vector.

Thus the SUM(Amount) for (July AND Fruit) combination is $= S_{R5\_200}+ S_{R5\_800}+ S_{R5\_500}+ S_{R5\_600}+ S_{R5\_700}+ S_{R5\_100}+ S_{R5\_400}+ S_{R5\_300}$

$= 000+800+500+1200+700+000+000+000$

$= 3200$

The SUM(Amount) for (July AND Fruit) is 3200 which satisfy threshold condition given in query 2. Therefore (July AND Fruit) combination is the part of final iceberg query 2 result.

**F.R6:(July AND Vegetable)**

The SUM(Amount) for (July AND Vegetable) is calculated by performing bitwise AND between R6 and Amount subset. The bitwise AND between (R6 AND 200) is performed as shown in Table 2.4. $S_{R6\_200}$ contain all zero. Therefore $S_{R6\_200}=000$. Similarly the bitwise AND operation between (R6 AND 800) is performed. $S_{R6\_800}$ contain 1’s only one time. So, $S_{R6\_800}= 800$.

**NOTE:** Here we can find that there is no need to repeat the bitwise operation between R6 and remaining subset of the amount because number of 1’s present in R6 is only 1. In 2nd step, we found the 1’s value as 800. But traditional method performs all operations and result in more fruitless operations.

Thus the SUM(Amount) for (July AND Vegetable) combination is $= S_{R6\_200}+ S_{R6\_800}+ S_{R6\_500}+ S_{R6\_600}+ S_{R6\_700}+ S_{R6\_100}+ S_{R6\_400}+ S_{R6\_300}$

$= 000+800+000+000+000+000+000$

$=800$
The SUM(Amount) for (July AND Vegetable) is 800 which does not satisfy threshold condition given in query 2. Therefore (July AND Vegetable) combination is not the part of final iceberg query 2 result.

**G. R7: August AND Milk**
The Sum(Amount) for (August AND Milk) is computed by performing Bitwise AND operation between R7 and Amount Subset.

The bitwise AND between (R7 AND 200) is performed which is as indicated in Table 2.4. S_R7_200 contain 1’s only one time. Therefore, S_R7_200= 200. Similarly remaining bitwise AND operations are performed and Sum value is calculated.

Thus the SUM(Amount) for (August AND Milk) combination is = S_R7_200 + S_R7_800 + S_R7_500 + S_R7_600 + S_R7_700 + S_R7_100 + S_R7_400 + S_R7_300

= 200+000+000+000+000+100+400+300

= 1000

The SUM(Amount) for (August AND Milk) is 1000 which does not satisfy threshold condition given in query 2. Therefore (August AND Milk) combination is not the part of final iceberg query 2 result.

**H. R8: August AND Fruit**
The combination (August AND Fruit) does not exist in the dataset. Therefore, there is no need to perform any operation on such a combination. Traditional algorithms do not take care of this aspect which results in fruitless operations.
I.R9: August AND Vegetable

The SUM(Amount) for (August AND Vegetable) is calculated by performing bitwise AND between R9 and Amount subset. The bitwise AND operation between (R9 AND 200) is performed which is as shown in Table 2.4. The result S_R9_200 contain 1’s only one time. Therefore, S_R9_200=200. Similar operations are performed between remaining combinations to find SUM(Amount) for (August AND Vegetable) combination.

**NOTE:** As R9 contains four 1’s, till the combination (R9 AND 100) summation of all four combinations is completed therefore further, AND operations can be skipped. However, traditional approach continues till the end of the sequence.

Thus the SUM(Amount) for (August AND Vegetable) combination is:

\[
S_{R9\_200} + S_{R9\_800} + S_{R9\_500} + S_{R9\_600} + S_{R9\_700} + S_{R9\_100} + S_{R9\_400} + S_{R9\_300}
\]

\[
= 200 + 000 + 000 + 600 + 700 + 100 + 000 + 000
\]

\[
= 1600
\]

The SUM(Amount) for (August AND Vegetable) is 1600 which satisfy threshold condition given in query 2. Therefore (August AND Vegetable) combination is the part of final iceberg query 2 result.

2.5.3 *Query with AVG Aggregate Function*

**Query 3:** Select Month, Category, AVG(Amount) from Monthly_Exp_25 group by Month, Category having AVG(Amount)>600;

In this case, the aggregate function is AVERAGE. It requires the complete procedure of SUM aggregate function. Then divide the summation by count of combination of attribute contributed in computing summation. The result of AVERAGE function passes through threshold check. The combinations which
satisfy threshold condition are included in final Iceberg query result. Final query 3 result is generated by performing bitwise AND between all combinations from R1 to R9 with 200,800,500,600,700,100,400 and 300.

Sum value of each subset is computed independently. It requires the Query1 as well as Query2 steps as described in section 2.5.1 and 2.5.2. The AVG value of R1: June AND Milk is as shown below.

\[
\text{SUM(Amount) for (June AND Milk) combination is } = S_{R1_{200}} + S_{R1_{800}} + S_{R1_{500}} + S_{R1_{600}} + S_{R1_{700}} + S_{R1_{100}} + S_{R1_{400}} + S_{R1_{300}} \\
= 400 + 000 + 500 + 000 + 000 + 100 + 000 + 300 \\
= 1300 \\
\text{AVERAGE(Amount) } = \frac{1300}{5} = 260
\]

The Average(Amount) for (June AND Milk) is 260 which does not satisfy threshold condition given in query 3. Therefore (June AND Milk) combination will not be the part of final iceberg query 3 result. Similar steps are followed to find AVG value of R2, R3, R4, R5, R6, R7, R8, and R9.

2.5.4 **Query with MIN Aggregate Function**

**Query 4:** Select Month, Category, MIN (Amount) from Monthly_Exp_25 group by Month, Category having MIN (Amount) > 200;

To evaluate MIN aggregate function of query each combination of Month and Category has to be check with Amount attribute. In this case, MIN (Amount) check is a threshold value in query 4. Query 2 evaluation plan applies to this aggregate function. Instead of performing SUM here MIN value check has performed.
2.5.5 **Query with MAX Aggregate Function**

**Query 5**: Select Month, Category, MAX (Amount) from Monthly_Exp_25 group by Month, Category having MAX (Amount)>500;

MAX aggregation function requires same steps as that of query 4. Only condition check is different. Threshold check is performed on MAX (Amount) which is a threshold value specified in query 5. As per the process of query 2 evaluation, the combinations which are the part of query 5 result can be found.

The summary of bitwise AND operation performed by Query 1 to 5 is as shown in Table 2.5. It is observed that more than 50% bitwise operations performed by traditional bitmap index approach are fruitless. This approach repeatedly access the bitmap index which is also a time-consuming process. This process become tedious and complex for large database. This increases the query processing time as well as database access time. This strategy for evaluating iceberg query is expensive and time-consuming. To overcome these limitations, this research work comes up with bitmap preprocessing strategy to evaluate the iceberg query. The detail of the preprocessing strategy is described in further chapters.

<table>
<thead>
<tr>
<th>Query and Aggregate Function</th>
<th>Total Bitwise AND operation Performed</th>
<th>Fruitful AND Operation</th>
<th>Fruitless AND Operation</th>
<th>AND Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.COUNT</td>
<td>9</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>2.SUM</td>
<td>81</td>
<td>44</td>
<td>37</td>
<td></td>
</tr>
<tr>
<td>3.AVERAGE</td>
<td>81</td>
<td>51</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>4.MIN</td>
<td>81</td>
<td>29</td>
<td>52</td>
<td></td>
</tr>
<tr>
<td>5.MAX</td>
<td>81</td>
<td>38</td>
<td>43</td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>333</strong></td>
<td><strong>166</strong></td>
<td><strong>167</strong></td>
<td></td>
</tr>
</tbody>
</table>
2.6 Summary

This chapter has reviewed the research work done on iceberg query processing technique, aggregation functions and bitmap indexing techniques. This chapter also reviewed the research related to the suitability of bitmap indexing technique in bitmap preprocessing strategy. The evaluation of iceberg query with different previous approaches degrades the performance of query as they require a large number of table scan. Iceberg query complexity is proportionally increased with database size. In case of a large database, traditional indexing technique takes a long time to access data and answer the Iceberg query. Bitmap indexing is suitable for large size, static and reads only databases.

This research intends to analyze large, historic and not frequently updated data set. In such a situation the bitmap indexing is the appropriate choice. In this way, this literature review helps to find suitability of bitmap indexing and cache for iceberg query processing. Demonstration of Iceberg query evaluation using existing method is included in this chapter. The base of evaluation is bitwise AND operation only.