A Redundancy-free IFC Storage Platform For Multi-model Scenarios based on Block Hash

Li S.[1], Gao G.[1,2,*], Wang W.[3], Liu H.[1], Zhu S.[1], Gu M.[1,2]
[1]School of Software, Tsinghua University, Beijing, China, [2]Beijing National Research Center for Information Science and Technology(BNRist), Tsinghua University, Beijing, China, [3]State Key Laboratory of Rail Transit Engineering Informatization (FSDI), China
gaoge@tsinghua.edu.cn

Abstract. This paper proposes a block hash (BH) method for efficiently storing multiple Industry Foundation Classes (IFC) models in the database. We find that there are lots of duplications when we store different models or different versions of a model into the database because of the temporal-spatial correlation of data between these models. The main idea of our approach is to divide these model files into appropriate blocks and to calculate the hash values of these blocks to reuse the them in different models. These blocks should not be too small in case too many nodes need to be compared, nor should they be too large in case it is difficult to find identical blocks to share between different models. So the BH method is proposed to efficiently make the database redundancy-free. For the experiments we use multi-versions of multiple models of a same project to validate this method. The experimental results show that our method is practicable and efficient.

1. Introduction

Building Information Modeling (BIM) technology has been widely applied in the construction industry. Since every stage of a facility will involves multiple institutions, data resource sharing becomes significant and urgent. BuildingSMART proposed Industry Foundation Classes (IFC), a standardized, digital description of the build asset industry. It provides an international standard for many different applications and promotes the exchange and sharing of data. As a way to exchange a BIM file between different software, IFC is designed to be non-redundant and concise in a monolithic file. However, with the application of BIM becoming more and more collaborative, and the model size becoming increasingly larger, non-file based BIM sharing platform is becoming necessary. These platforms, such as CDE (Common Data Environment), FM (Facility Management platform) and Archiving, etc., usually provide storing, querying, managing and viewing functions for multiple BIM models supported by database.

Typically, many of these models may be different parts of a same project, or different versions of a same design, or generated by the same team, so the data in these models may have temporal or spatial correlation. The well-designed inner structure of the IFC file cannot prevent the large amount of duplicate data across these models.

So many people try to extract IFC model information and store it in different types of databases. Nevertheless, the existing approaches seem inefficient for some multi-model task scenarios.

In this work, we proposed the block hash (BH) method to merge the duplicate nodes in IFC files. This method takes advantage of the structure of IFC. It not only reduces the space cost of the storage, but also reduces the upload time required for the storage. Based on this method, we have established a redundancy-free IFC storage platform. In the platform we provide the interfaces for saving and querying information from multiple models. we adopt MongoDB for the underlying database of the platform for its good applicability and great read performance in the case of high load. Moreover, the block hash method may also be used for storage based on other databases. The current experimental results show that BH method is very effective in handling with multi-model tasks. And it also has good scalability for subsequent research.

https://doi.org/10.7146/aul.455.c198

2. Related Works

2.1 Model Storage in BIM

IFC has many expression formats like the SPF (STEP Physical File) format, XML, JSON and so on. The SPF format includes HEADER and DATA two parts. HEADER is used to record some meta data such as IFC Schema version, model name and description etc. DATA is for the real model data which is organized according to the IFC specification.

Krijnen and Beetz (Krijnen and Beetz, 2016) put forward the method of using HDF5 to store IFC models, which greatly reduces the querying time of some components such as obtaining the largest window in the model. This method is more advantageous than the traditional formats in many aspects. But there are still difficulties in data sharing and updating.

These file-based methods mentioned above are still unstructured or semi-structured. This makes it difficult to realize the data features urgently needed by BIM, such as data management, sharing and updating. Therefore, more people are exploring other storage methods like using databases or blockchain to store BIM model to solve these problems.

BuildingSMART is carrying out the experiment of using SQLite to store IFC model. The purpose is to provide a standard format of SQLite to store IFC data. In the aspect of relational database, Li et al. stored IFC model into ORACLE database in order to verify the feasibility of IFC database storage idea (Li et al., 2016). The experiment proved the feasibility of lossless storage of IFC data by database, but the actual speeds were not satisfactory. Beetz et al. proposed BIM Server of BerkeleyDB which is a key-value database (Beetz et al., 2010). The architecture realized the conversion from IFC file to database by building a service layer and the KeyValueStore Interface was provided to connect to different databases. But so far, only the open source BerkeleyDB Java Edition database is realized (opensourceBIM, 2021). Yuan et al. also adopted the column-oriented database HBase to store IFC model (Yuan and Shihua, 2017). Jiang and Wu put forward the method of storing IFC by using Elastic Search framework and the graph database Neo4j. However only the spatial data of IFC model was stored (Jiang and Wu, 2018). In 2019, Gao et al. proposed to use knowledge base and graph database to store IFC model data. This method was mainly used to improve the efficiency of automatic model checking (Gao et al., 2019).

Apart from the methods mentioned above, there are some storage systems such as the BIM-server (Singh, Gu and Wang, 2011). These achievements indicate that structured storage of BIM data is promising. The purpose of this research is to provide a general hashing method to effectively compare the differences between different models or versions. It is helpful to reduce the duplicate data and get the difference easily.

2.2 Model Compression in BIM

IFC has excellent structural design, which can reuse data to reduce storage space. However due to the different IFC export algorithms adopted by different applications, there will be some duplicate nodes in one model. Especially some fundament types like IfcCartesianPoint and IfcPropertySingleValue will have some duplicate instances. There are some compression algorithms to remove these like the IFCCompressor which compresses the IFC files by line by line (Sun et al., 2015) and ACC4IFC which takes the default value into consideration (Du et al., 2020). The basic ideas of these algorithms are very concise. They want to remove the duplicate nodes and reuse the left one to make the model as small as possible. And the facts have proved that these methods can effectively compress IFC models. These algorithms mainly focus on the compression of single model. But the more important problem that this research intends to solve

is the duplicate data between different models especially different versions of the same model like Figure 1. Although we can still use similar methods to compress these models but there will be some problems and we will discuss in the methodology and experiment section.

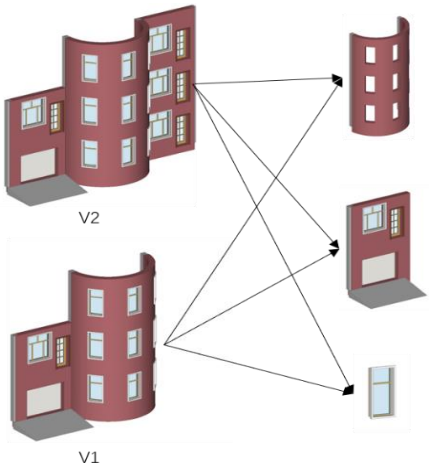

Figure 1: The elements shared in different model versions.

In addition to the content-based compression methods, there are also some other compression algorithms like the mesh simplification (Algorri et al., 1996; Cignoni et al., 1998) and so on. Most of these methods simplify the geometric information of the model and will lose some fine geometric information. In this study we want to keep all the information in the origin model and we will not use these lossy compression methods.

3. Research Methodology

In this section, we mainly present the methodology how we implement hashing algorithm and the way we build our platform based on BH.

Every release of the IFC specifications has strict regulations on every node type. The inheritance diagram (or specification) of IfcRoot in IFC2X3 is presented below.

ENTITY IfcRoot

ABSTRACT SUPERTYPE OF (**ONEOF** (IfcPropertyDefinition, IfcRelationship, IfcObjectDefinition));

GlobalId	:	IfcGloballyUniqueId;
OwnerHistory	:	IfcOwnerHistory;
Name	:	**OPTIONAL** IfcLabel;
Description	:	**OPTIONAL** IfcText;

UNIQUE

UR1	:	GlobalId;

END_ENTITY;

It's clear to figure out every parameter in this basic type. There are some reference parameters like the OwnerHistory. According to the specification we can define one IfcOwnerHistory node here or reference another node whose type is IfcOwnerHistory.

The size of one node is determined by these parameters. Assuming that the average size of one certain type is s and the file size is S, we can get that:

$$S = \sum_{t \in types} s_t \times c_t \quad (1)$$

where the c_t means the number of the node whose type is t.

It can be seen from Equation (1) that in order to reduce the storage space of IFC models, we can reduce the size of a single node or the number of nodes. It is obvious that we can use one algorithm similar to the content-based compression algorithms to compare these models as one to remove the duplicate nodes. However, the number of nodes is too large to deal with and the different content makes it hard to compare these nodes one by one. The hash algorithm is used to convert an arbitrary node into an encrypted output of a fixed length and it helps to decrease the complexity of comparison. Our platform adopts MD5 (Rivest and Dusse, 1992) as the hash algorithm, so each hash is 16bytes long. The number of nodes in a 200MB IFC file is about 3000000, which means that about 45.78MB of hash data needs to be saved.

If we get rid of the hash method, although we don't need to save these hashes, the comparison between different lengths of content will become more difficult. Each new model uploading will need to load all the models which is impossible for the current hardware. So, it is necessary to reduce the number of hashes, that is, the number of nodes, in order to speed up the calculation. For the above reasons it's important to merge some nodes as one whole part to participate in the procession.

3.1 Calculate one block from IFC model

Because of the reference relationship in IFC structure, the algorithm can start from one node that is not referenced by other nodes, and recursively find all the referenced nodes to form a data block. However, this will cause a lot of duplicate nodes. For example, there is usually only one IfcOwnerHistory node in one IFC model and all other nodes that inherit from IfcRoot reference to this one. And the above blocking strategy will cause this node to be repeated tens of thousands of times, resulting in a sharp increase in storage space. So, in this circumstance, the nodes with many references should also be partitioned and the equation of S becomes:

$$S = s_t + \sum_{t' \in refs} B_{s_{t'}} \times Q_{t'} \tag{2}$$

$$S \approx \sum_{t' \in types} B_{s_t} \times c_t \times (1 - Q_t) \tag{3}$$

where B_{s_t} means the block starts with the node whose type is t, s_t means the average size of node whose type is t, $refs$ means other node types referenced by type t, and Q_t represents whether the node is the head node, which can be expressed as Equation (4).

$$Q_t = \begin{cases} 0, & t \text{ is the head type} \\ 1, & t \text{ is not head type} \end{cases} \tag{4}$$

If every type is the head type, the Equation (3) degenerates to Equation (1) which means calculating hash for each line.

According to the above analysis, it can be concluded that in order to reduce the data expansion, the key is to define the head type. Therefore, the following BH method is proposed.

1. Considering that the header nodes are easy to create index, all types with GlobalId need to be used as header type.
2. Count the quantities of nodes corresponding to each type of all IFC models in this project and save them.
3. Start from the nodes with GlobalId to find the referenced nodes recursively. The number of referenced nodes corresponding to each type in the recursion process is counted and saved.

4. Divide the referenced number of this type by the number of nodes, and set a threshold. When the result is greater than this threshold, the Q value of this type is set to 0.

The change of the threshold will result in the following results: If the threshold value is too high, most types will not be partitioned. It will result in excessive data redundancy. If the threshold value is too low, it will lead to a large number of types. In extreme cases, it will degenerate into Equation (1). Therefore, the factors to be considered when setting the threshold include the configuration of the server for storage, the total model size of the project, the network situation for uploading and the general configuration of the computer hardware for uploading IFC files, etc.

Since the project models may be constantly increasing from the beginning of the project, the threshold cannot be accurately calculated. Considering the various factors that affect the storage space, querying efficiency and processing speed, we can finally roughly estimate a threshold range according to the required hash quantity and storage space. If the total number of nodes is in the order of tens of millions, it is hoped that the number of hashes can be reduced to the previous 1/4. In this algorithm, the threshold can be selected in the range of 4-6.

3.2 Calculate hash for each block.

After we get the blocks of the model, we require removing the duplicate blocks. Calculating the hash for each block is necessary. For there is only one header in each block, we can replace the reference with the node itself in this header. In this way we can get one node that combines all nodes in this block like the nodes #13 and #14 in Figure 2 which replace the reference in #7 and # 12. And the final block is only related to the existing blocks like #14. It should be noted that the existing blocks not only refer to the blocks in this model, but also include the blocks in the other uploaded models. Therefore, the line number cannot be duplicated and must be incremented even in different models. Then we will calculate hash for #13 and #14 using the text. Although the hashing algorithms are sensitive to changes like adding or removing just one whitespace. We will pay attention to the IFC models derived from other applications like Revit for it's hard to modify the IFC model itself currently. So, the format of IFC is fixed and we can ignore some small changes.

```
DATA;
#1= IFCPERSON($,'\X2\672A5B9A4E49\X0\',$,$,$,$,$,$);
#3= IFCORGANIZATION($,'\X2\672A5B9A4E49\X0\',$,$,$);
#7= IFCPERSONANDORGANIZATION(#1,#3,$);
#10= IFCORGANIZATION('GS','GRAPHISOFT','GRAPHISOFT',$,$);
#11= IFCAPPLICATION(#10,'22.0.0','ARCHICAD-64','IFC add-on version: 4005 CHI FULL');
#12= IFCOWNERHISTORY(#7,#11,$,.ADDED.,$,$,$,1557220048);

#13= IFCPERSONANDORGANIZATION(IFCPERSON($,'\X2\672A5B9A4E49\X0\',$,$,$,$,$,$),IFCORGANIZATION($,'\X2\672A5B9A4E49\X0\',$,$,$),$);
#14= IFCOWNERHISTORY(#13, IFCAPPLICATION(IFCORGANIZATION('GS','GRAPHISOFT','GRAPHISOFT',$,$);
                    ,'22.0.0','ARCHICAD-64','IFC add-on version: 4005 CHI FULL');,$,.ADDED.,$,$,$,1557220048);
```

Figure 2: Example to Calculate BH.

3.3 Hash comparison based on the node block

The way to store and query all hash data has a strong impact on the upload time. Here we use the Redis and bloomfilter (Bloom, 1970) to store and query hash data. To accelerate this process, we also add the local cache to save the known results. When querying whether this value exists, first we need to look for it in local hash cache, next the local bloomfilter and finally the Redis until we get the precious consequence. The process can be shown as Figure 3.
The bloomfilter essentially holds an array of bits of length m. K (k<m) positions of each data are calculated by k hash functions, and the corresponding position is set to 1 in the array. The

structure can be shown as Figure 4. When querying, the data is also calculated by these k hash functions to get the corresponding positions. If the corresponding positions are all 1, it is proved that the data may exist. If one of them is 0, the data must not exist. It is originally a plug-in in Redis. We will load these data into one local self-developed bloomfilter before the upload starts to reduce the impact of network delay.

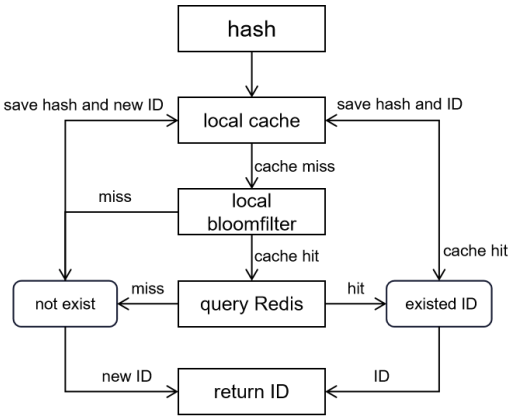

Figure 3: Hash Querying Process.

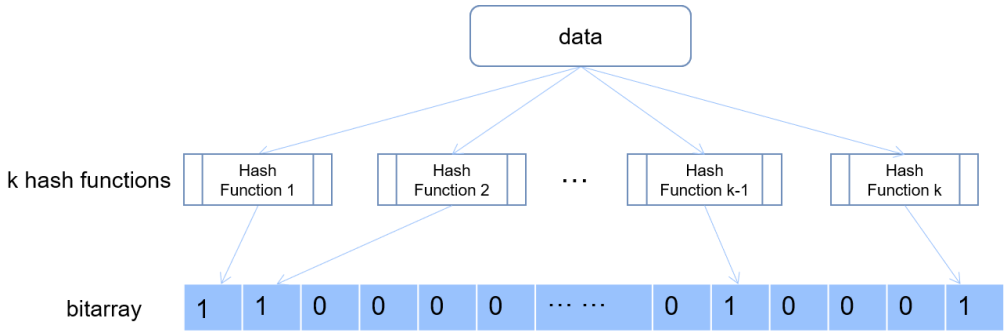

Figure 4: Bloomfilter Structure.

3.4 Creating block index in the database

After the model is uploaded, the index that indicates the reference between different nodes and some other indexes will be constructed, which will also include composite index. All subsequent query processes are based on these indexes. These indexes include the GlobalId index, the model index, the reference blocks index etc. According to the document of MongoDB, for a compound multikey index, each indexed document can have at most one indexed field whose value is an array. So here we can create as more as possible compound indexes apart from those who have two fields whose value is an array to help the data querying.

4. Experiments and Results

4.1 Test environment and models

First of all, we need to set up our platform. It includes MongoDB and Redis two databases. To reduce network delay, we recommend to deploy them on one server or two servers located in the same high-speed LAN (1GbE we used in our experiments). We chose the latter in order to present the experimental results better. The following experiments were performed on two

laptop computers both equipped with an Intel Core i7-10700F CPU (2.9 GHz), 32GB RAM and 128G hard disk.

For the experimental models we selected three models from different floors of the same building. We removed parts of these models and got another three models that we called the second version (V2). Figure 5 presents the first version (V1) and V2 of these models. There are so many similar elements like pipelines, walls between different models or different versions. All the evaluations were performed on the six models. The sizes of the six models are 353MB, 360MB, 261MB, 196MB, 11.4MB and 10.8MB from a to f in Figure 5. We used three methods to upload the models, namely BH, line hash (LH) which means calculating hash line by line and no hash (NH) which means just storing every line of the IFC models. Two experiments were carried out for each method and tested removal of the spatial and temporal redundancy respectively. Because we will not change the BH strategy in the whole experiment, so the results are independent of the order of upload, which represents the addition or deletion of some parts. As for the modification, we can regard it as deletion and then addition. So, these experiments are enough to cover the common operations in the real-world version.

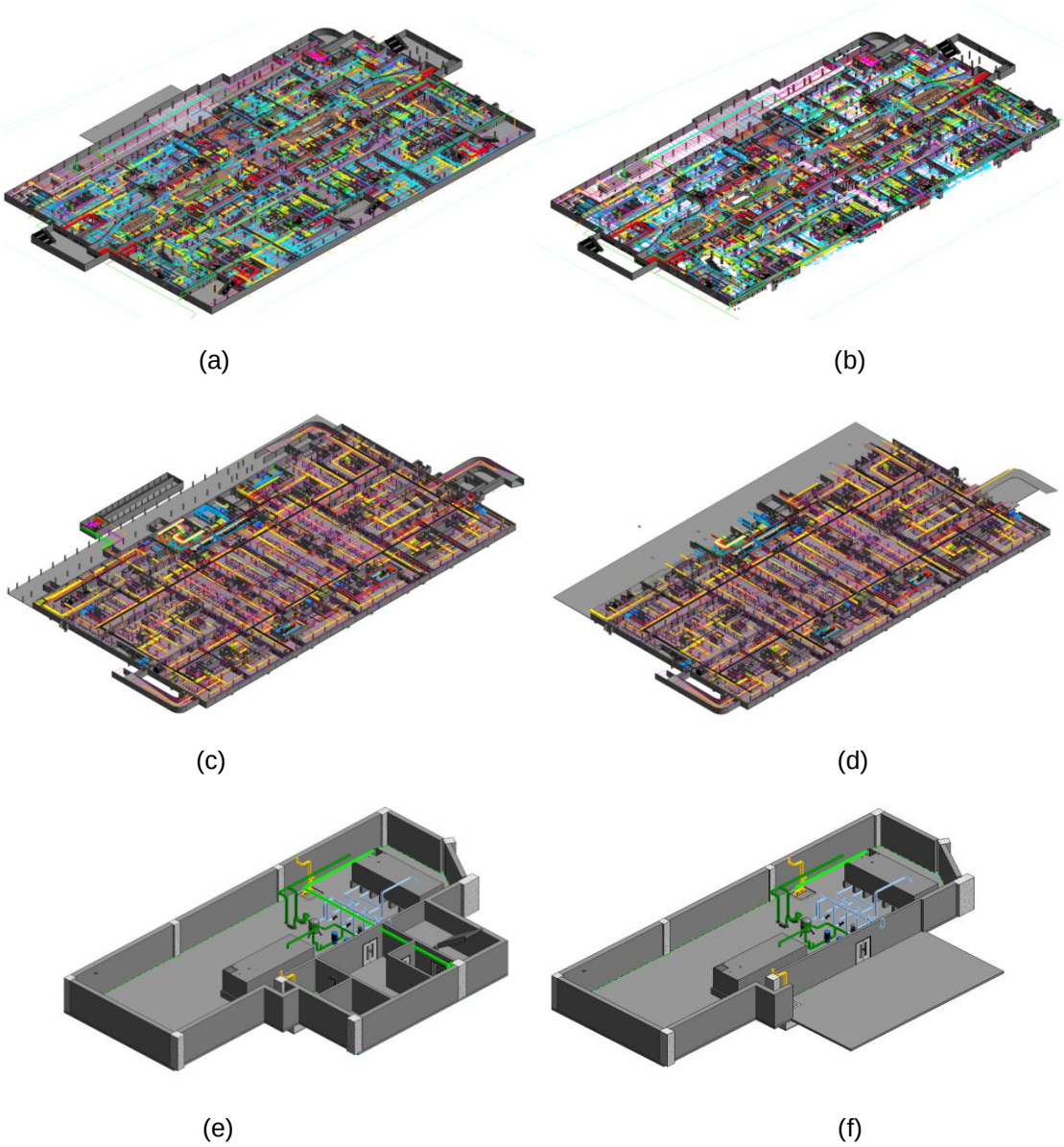

Figure 5: Test models. These models are from the basements of the same building. (a), (c), (e) are V1 models and (b), (d), (f) are V2 models.

4.2 Evaluation of removing structural redundancy.

We uploaded all the V1 models to test these methods and the Table 1 shows the upload results.

Table 1: Upload V1 models results.

Upload method	Time (m)	MongoDB space (MB)	No. of Block or Line	Redis Mem (MB)	Mem (GB)
BH	20	788.7 + 400.6	2899510	220.72	5.1
LH	132	653.6 + 682.2	6577498	489.17	5.3
NH	120	993.5+ 940.4	10578153	0	9.3

The occupied MongoDB space is divided into two parts because we need to build some index for querying. In order to be more intuitive, we show this part of data separately. The first part is the model data and the second is the index data. The index space is positively correlated with the number of hashes because we need to build index for every node.

Comparing the results of LH and NH, we can find that there are so many duplicate data between different models. The method LH can effectively remove these duplicate data but because of too many hashes, the speed, occupied Redis memory and index space are not satisfactory. By BH, we reduced the number of hashes to 27% of NH and 44% of LH. Although the space occupied by model data of BH is bigger than LH, the index takes up much less space than the other two methods. Due to less hashes, BH is six times faster than the other two methods and Redis takes less memory. Too many nodes will cause frequent reading and writing of a small amount of data for MongoDB and Redis, which will greatly affect the upload speed. The maximum memory occupied by the program during running is also reduced.

4.3 Evaluation of removing version redundancy.

We uploaded the V2 models on the basis of experiment 4.2 and get the final results which are show in Table 2. Except for the time and memory, all the remaining records are the sum of two experiments.

Table 2: Upload V2 models results.

Upload method	Time (m)	MongoDB space (MB)	No. of Block or Line	Redis Mem (MB)	Mem (GB)
BH-V1	20	788.7 + 400.6	2899510	220.72	5.1
BH-V2	15	1.0 GB + 696	3772968	286.05	4.8
LH-V1	132	653.6 + 682.2	6577498	489.17	5.3
LH-V2	98	826.1 +1.2GB	7455914	566.72	3.2
NH-V1	120	993.5+ 940.4	10578153	0	9.3
NH-V2	102	1.9GB+1.78GB	20090161	0	9.1

According to the NH results, the line number of V2 is almost the same as V1. In this experiment the method LH removed more nodes and the proportion of remaining nodes in the total decreased from 62% to 37%. In the method BH, the number of blocks is just about 1.3 times of V1. All these indicate that these two methods can remove lots of redundant data between model versions. It is still the number of nodes that causes the other two methods to be slow. Although there is no hash in NH means we don’t need to calculate, save and query the hash, too many nodes will still cause slow writing to MongoDB and large memory consumption. At the same

time the index space is also a factor that cannot be ignored. There are so many nodes we will never query or get. It's a waste to build index for these nodes. If we take the index space into consideration, BH's space occupation will become the lowest. The other advantages of BH are the same as experiment 4.2.

5. Conclusion

In this work, we have implemented the redundancy-free IFC storage platform which is based on the BH method. Compared with the other method like the traditional storage ways or the other hash methods, this approach has the following advantages:

- There is a lot of redundancy between different models or different versions. BH can help to remove these duplicate data and save space.
- Compared with other method, BH can greatly improve efficiency by reducing the number of hashes. This method can reduce the reading and writing times of the database, and organize IFC data more efficiently.
- The memory required for BH operation and Redis is lower which makes it better used in practical application.

By BH and other improvements we built the redundancy-free IFC storage platform. The advantages of this platform largely match the current BIM development trend-fast data sharing and exchange. However, due to the huge amount of data, storage based on database cannot completely replace file-based storage at present. In this study, MongoDB, a non-relational database is used to store IFC model data, and some algorithms such as BH and hash querying acceleration are proposed by using the structural characteristics of IFC, which greatly improves the usability of BIM database.

This proposed method has been successfully approved that BH is effective. But there are still some problems to be solved, such as how to reach the optimality, build more efficient indexes or make the database support more IFC data. The future research may explore how to modify the current database technology and make it more suitable for the massive, rapid-changing BIM data.

Acknowledgement

This work was supported by the State Key Laboratory of Rail Transit Engineering Informatization (FSDI) "Research on the framework of Enterprise Railway BIM collaborative platform" and the 2019 MIIT Industrial Internet Innovation and Development Project "BIM Software Industry Standardization and Public Service Platform". The conclusions herein are those of the authors and do not necessarily reflect the views of the sponsoring agency.

References

Algorri, M.E. and Schmitt, F. (1996, August). Mesh simplification. In Computer Graphics Forum (Vol. 15, No. 3, pp. 77-86). Edinburgh, UK: Blackwell Science Ltd.

Beetz, J., van Berlo, L., de Laat, R., & van den Helm, P. (2010, November). BIMserver. org–An open source IFC model server. In Proceedings of the CIP W78 conference (p. 8).

Bloom, B.H., (1970). Space/time trade-offs in hash coding with allowable errors. Communications of the ACM, 13(7), pp.422-426.

Cignoni, P., Montani, C. and Scopigno, R. (1998). A comparison of mesh simplification algorithms. Computers & Graphics, 22(1), pp.37-54.

Du, X., Gu, Y., Yang, N., & Yang, F. (2020). IFC File Content Compression Based on Reference Relationships. Journal of Computing in Civil Engineering, 34(3), 04020012.

Jiang, S., Wu, Z. (2018). Research on Cloud Storage and Retrieval Method of BIM Spatial Relational Data. Journal of Graphics, 39(5), p.835.

Gao, G., Zhang, Y., Liu, H., Li, Z., Gu, M. (2019). Research on IFC Model Checking Method Based on Knowledge Base. Journal of Graphics, 40(6), p.1099.

Krijnen, T., & Beetz, J. (2016). Efficient binary serialization of IFC models using HDF5. In Proceedings of the 16th International Conference on Computing in Civil and Building Engineering (ICCCBE2016) Osaka, Japan.

Li, H., Liu, H., Liu, Y., & Wang, Y. (2016). AN OBJECT-RELATIONAL IFC STORAGE MODEL BASED ON ORACLE DATABASE. International Archives of the Photogrammetry, Remote Sensing & Spatial Information Sciences, 41.

opensourceBIM, (2021). Database Versioning · opensouceBIM/BIMserver Wiki · Github, https://github.com/opensourceBIM/BIMserver/wiki/Database---Versioning, accessed January 2022.

Rivest, R. and Dusse, S., (1992). The MD5 message-digest algorithm.

Singh, V., Gu, N., & Wang, X. (2011). A theoretical framework of a BIM-based multi-disciplinary collaboration platform. Automation in construction, 20(2), 134-144.

Sun, J., Liu, Y. S., Gao, G., & Han, X. G. (2015). IFCCompressor: A content-based compression algorithm for optimizing Industry Foundation Classes files. Automation in Construction, 50, 1-15.

Yuan, C., & Shihua, Y. (2017). Research on HBase-based BIM Model Storage Technology. Journal of Information Technologyin Civil Engineering and Architecture, 9(4), 74-81.