Report on Batch Codes and Their Applications

Anand Mayuresh Vivekanand
Supervisor: Dr. Vitaly Skachek

Abstract

Batch codes were proposed for use in the following applications: trading maximal load for storage in certain load-balancing scenarios, and amortizing the computational cost of private information retrieval(PIR) and related cryptographic protocols. In this report we present some constructions of the batch codes as described in [1].

1 Introduction

In the paper [1] Ishai et al. introduced a novel idea of constructing a code (i.e. batch code) to minimize the worst-case maximal load on the devices. In this scenario we are faced with the problem of optimizing the worst-case maximal load and the total information storage size. The tradeoff between these two parameters motivates the construction of batch codes.

The problem : "We are given a large database of n items which has to be distributed on m devices, where items corresponds to bits and devices to servers or disk respectively. After we distribute the given database[1] a user wants to retrieve an arbitrary subset of k items from the database by reading the the data stored on the various devices. Our goal is to minimize the worst-case maximal load measured in number of bits read on any of the m devices while trying to minimize the total amount of space used for data storage."

Example 1: Consider the above problem for the parameter $m = 3$. One naive way to reduce the load can be to store the whole database of n items on each of the three devices and retrieve the k items by querying for $k/3$ items from each device. Thus, we have the following parameters:

- Total information storage size $= 3n$
- Maximum size of query $= \lceil \frac{k}{3} \rceil$

This solution increases the storage by three times which may not be feasible for large databases.

Example 2: Now if we can cope up with the increase in total storage by an amount of 50%. In this case we can cleverly partition the original database into two parts say, L and R. We then store L on the device m_1, R on device m_2 and $L \oplus R$ on device m_3. In order to retrieve the bits i_1 and i_2 we have two cases:

[1] the bits of information

1. i_1 and i_2 are located on the same part (*wlog* say m_1). We can retrieve them first by querying i_1 from m_1 and simultaneously querying i_2 from m_2 and $i_1 \oplus i_2$ from m_3. We then XOR the outputs to get the respective bits.

2. i_1 and i_2 are on different parts. We can now query both the devices simultaneously to get the respective bits.

Thus, we have the following parameters:

- Total size of information Storage $= 1.5n$
- Maximum size of query load $= \lceil \frac{k}{2} \rceil$

Hence, we have achieved a reduction of the storage from $3n$ in Example 1 to $1.5n$ in Example 2 with a moderate increase in maximal load. Here, we tradeoff size for maximal load

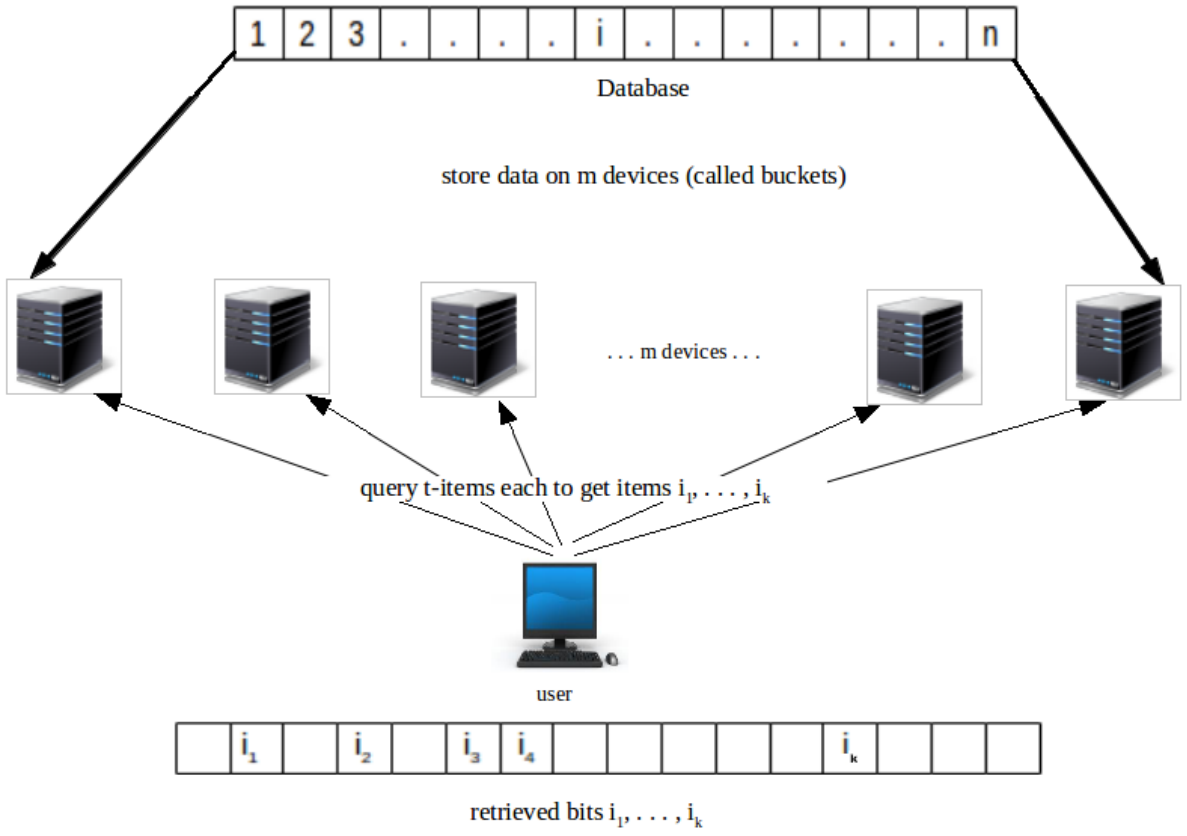

Figure 1: Use of batch codes to retrieve bits of data.

This problem can be described in terms of batch codes. An (n, N, k, m, t) batch code over an alphabet $\Sigma = \{0, 1\}$ encodes a string $x \in \Sigma^n$ into an m-tuple of strings $y_1, y_2, y_3, \ldots, y_m \in \Sigma^*$ (also referred to as buckets) of total length N where Σ^* denotes the set of all possible strings, such that for each k-tuple (batch) of indices $i_1, i_2, \ldots, i_k \in [n]$, the entries $x_{i_1}, x_{i_2}, \ldots, x_{i_k}$ can be decoded by reading at most t symbols from each bucket.

The bucket in the above definition corresponds to the devices, the encoding length N to

the total storage, and the parameter t to the maximal load. We will refer to n/N as the rate of the code.

Similarly to the above examples, we now have the challenge to simultaneously minimize the parameters N, m. For the practical purposes the alphabet Σ is taken to be the binary alphabet. Also, if we restrict $t = 1$, then it means that one can access the given devices only once[2]. This is what we will refer to as the "batch code" or (n, N, k, m) batch code. Thus, we require that $m \geq k$ in this default setting.

From the definition of the batch code we have two trivial construction of batch codes:

1. $\mathbf{C}(x) = (x, x, \ldots, x)$ *i.e.* here we replicate x in each bucket. Thus, we have the following parameter:
   - no. of buckets $m = k$ (optimal)
   - rate of code $= 1/k$ (very low)
2. $\mathbf{C}(x) = (x_1, x_2, x_3, \ldots, x_n)$ *i.e.* here we store each bit of x in a separate bucket. Thus we have the following parameters:
   - no. of buckets (*i.e.*) $m =$ size of the code (very large)
   - rate of the code $= 1$ (optimal)

Hence, as we can see, the two parameters are inversely proportional. We would therefore like to achieve optimal between these two parameters.

In the above scenarios we have considered that single user wants to retrieve a certain subset of the database. Practically, there can be many users, each wanting to retrieve a certain subsets of data (say, queries). This scenario has been modeled by multiset batch codes. In this scenario we have k distinct users, each holding some query i_j. Each item x_{i_j} should be recovered from the bits read by j^{th} user alone, rather than all the bits that were read. The k queries which were assumed to be different in the previous setting may not be true here. The indices i_j forms a multiset and code is called the multiset batch code.

2 Preliminaries and Definitions

Here we define the different variants of the batch codes and the related notions that would be essential to show the construction of batch codes.

Definition 1 (batch code)**.** *An (n, N, k, m, t) batch code over Σ is defined by an encoding function $\boldsymbol{C} : \Sigma^n \to (\Sigma^*)^m$ (each output of which is called a bucket) and a decoding algorithm $\boldsymbol{A}$ such that:*

- *The total length of all m buckets is N (where the length of each bucket is independent of x),*

[2]alternatively, only one symbol can be read from a device or bucket

- *For any $x \in \Sigma^n$ and $i_1, \ldots, i_k \in [n]$, $\boldsymbol{A}(\boldsymbol{C}(x), i_1, \ldots, i_k) = (x_{i_1}, \ldots, x_{i_k})$, and $\boldsymbol{A}$ probes at most t symbols from each bucket in $\boldsymbol{C}(x)$ (whose positions are determined by $i_1, \ldots, i_k$), where $[n] = \{1, 2, \ldots, n\}$.*

We will sometimes refer to x as the database. By default, we assume batch codes to be systematic, i.e., the encoding should contain each symbol of x in some fixed position. Finally, an (n, N, k, m) batch code is an $(n, N, k, n, 1)$ batch code over $\Sigma = \{0,1\}$.

Definition 2 (Multiset batch code)**.** *An (n, N, k, m) multiset batch code is an (n, N, k, m) batch code satisfying the following additional requirement. For any multiset $i_1, i_2, \ldots, i_k \in [n]$ there is a partition of the buckets into subsets $S_1, S_2, \ldots, S_k \subseteq [m]$, such that each item $x_{i_j}, j \in [k]$, can be recovered by reading (at most) one symbol from each bucket in S_j. This can be naturally generalized to $t > 1$.*

The following special case of (multiset) batch codes will be particularly useful:

Definition 3 (Primitive batch code)**.** *A primitive batch code is an (n, N, k, m) batch code in which each bucket contains single symbol, i.e. $N = m$.*

Note: the primitive batch code is trivial for the single user setting but non-trivial for the multiuser setting because an item can be queried by multiple users.

Lemma 1. *The following holds both for standard batch codes and for multiset batch codes:*

1. *An (n, N, k, m, t) batch code (for an arbitrary t) implies an (n, tN, k, tm) code (with $t = 1$).*
2. *An (n, N, k, m) batch code implies (n, N, tk, m, t) code and an $(n, N, k, \lfloor \frac{m}{t} \rfloor, t)$ code.*
3. *An (n, N, k, m) batch code implies an (n, N, k, m) code over $\Sigma = \{0, 1\}^w$, for an arbitrary w.*
4. *An (n, N, k, m) batch code over $\Sigma = \{0, 1\}^w$ implies a (wn, wN, k, wm) code over $\Sigma = \{0, 1\}$.*

Sketch of proof:

1. The (n, N, k, m, t) batch code should be able to query t bits in one go from each device. Hence, if restricted to just query 1 bit every device could be seen as being replicated t times so that t bits can be queried from it. Thus we have tm devices and thereby the total storage of N increase to tN.
2. We can see that initially the code can query only 1 bit from each device and if we want to increase this query to t bits for each device then we have to increase the total query by t times *i.e.* to tk. Now as the total access for device increases by t times so does the total number of queries.
3. Follows from the definition.
4. The size of single string which was earlier w now is read as of size one therefore the total size of the database should expand by the same amount *i.e.* w. This also increases the total storage and the devices requires by w times to wN and wm respectively, while keeping the amount of the bits to be retrieved (*i.e.* size of the batch) to be constant as k.

3 Construction of Batch Codes

In this section we present few batch codes constructions as presented in [1]. Simultaneously we also define some mathematical structures which would be needed to understand the constructions.

3.1 Batch codes from Unbalanced Expanders

Definition 4 (Graph). *A graph G is an ordered pair $(V(G), E(G))$ consisting of a set $V(G)$ of vertices and set $E(G)$, disjoint from $V(G)$, of edges, together with an incidence function ϕ_G that associates with each edge of G an unordered pair of (not necessarily distinct) vertices of G.*

Definition 5 (Matching). *A matching $MinE(G)$ is a set of pairwise non-adjacent edges (i.e.), no two edges share a common vertex.*

Definition 6 (Bipartite graph). *A graph is bipartite if the vertices can be partitioned into two sets, X and Y, so that the only edges of the graph are between the vertices in X and the vertices in Y.*

Theorem 1 (Hall's Theorem). *There is a matching of size $|A|$ if and only if every set $S \subseteq A$ of vertices is connected to at least $|S|$ vertices in B, where A and B are the two sets of a bipartite graph.*

We now try to construct the batch code using the bipartite graph. This code would be a replication based code, where each bit encoding is a physical bit of x. The code may be represented using n vertices on the left as the data bits (or items) and m vertices on the right which corresponds to the buckets (or devices). The edge joining the two vertices on the opposite sides suggests that the corresponding bit is contained in that bucket.

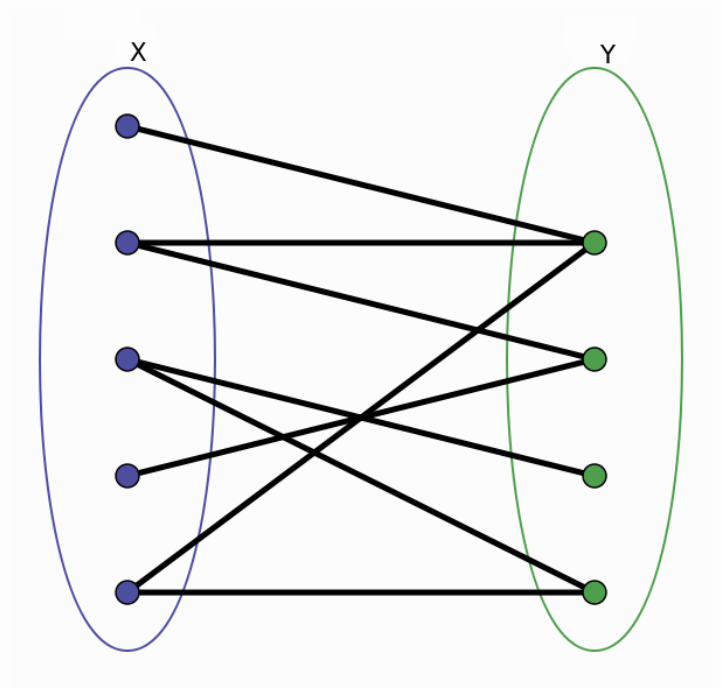

Figure 2: An example of a bipartite graph

Thus, by the Theorem 1, the given bipartite graph is an (n, N, k, m) batch code if and only if each set S of at most k vertices on the left has at least $|S|$ neighbours on the right.

Definition 7. *(Expander) A bipartite multigraph G is a (K, A) (vertex) expander if for all sets S of at most K vertices, the neighbourhood $N(S) = \{u \mid \exists\ v \in S\ s.t.\ (u, v) \in E\}$ is of size at least $A * |S|$*

Here we show how to construct the batch code from the unbalanced expanders.

Goal: To construct the bipartite graph for the given n bits and k bits to be retrieved.
Input: set X of vertices: n-bits data (or items), set Y of vertices: m - devices (or buckets) and repetitions: d.
output: k-bits of specific data subset (or batch)
Procedure:
for each vertex $u \in A$
repeat d times:
Choose an element $v \in B$, uniformly at random
if edge(u, v) does not exist
add an edge(u, v)
end.

Theorem 2. *Let $m \geq k(nk^{\frac{1}{d-1}})t$. Then, with probability at least $1 - t^{-2(d-1)}$, the neighbourhood of every set $S \subset A$ such that $|S| \leq k$ contains $|S|$ vertices in B.*

The expander based construction of batch codes has the following features

- In the case when only a single user can query, this approach is equivalent to the replication based approach. For a fixed good constant degree expander graph the encoding function can be computed in linear time. Also, for a change of a single bit of x only a constant number of bits in the encoding needs to be updated.
- For a constant d, the value of m depends on both k and n (as can be seen from Theorem 2)
- If we fix the value of k the expansion properties can be checked in polynomial time.
- By using the following parameters, (i.e.) $d = O((1/\epsilon)\log nk)$, we have $m = (1+\epsilon)k$. This is possible because of the very weak expansion reqirement.
- The constructions of unbalanced expanders from [5] yields $d = 2^{(\log\log n)^3}$ and $m = O(kd)$ while the construction from [6] yields two possible settings of parameters: $d = \log^c n$ for some constant $c > 1$, $m = 2^{(\log k)^{1+\epsilon}}$ and $d = 2^{(\log\log n)^2}$, and $m = k^c$, for some constant $c > 1$.

Expander-based batch code has rate not exceeding $\frac{1}{2}$.

3.2 Subcube Code

In Example 2 we had constructed a code for $m = 3$ with rate of $k/2$ and 50% increase in the storage size. We now carry this idea forward by recursively applying the same approach and partitioning each block into similar sub-blocks. This is shown in figure 3.

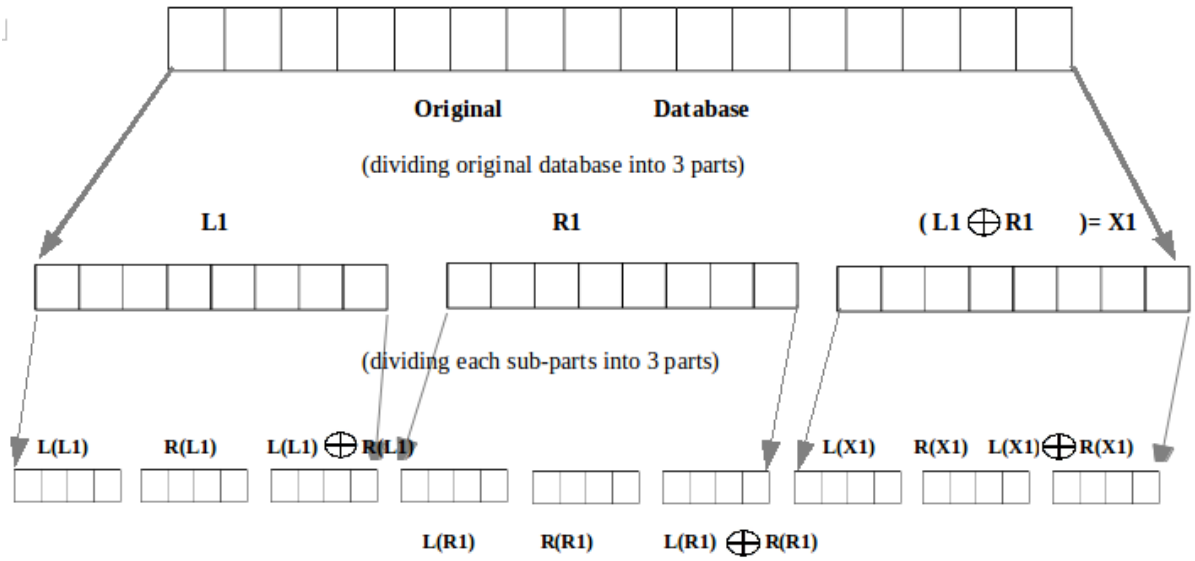

Figure 3: Recursive approach to construct Subcube Code

Thus, we can see that after the 2^{nd} level of recursion the size of the database that each bucket stores is $n/4$[3]. Also, the number of buckets increases to 9. Thus, now we have 9 buckets with total storage data of size $9n/4$ as compared to originally 3 buckets with data $3n/2$. But now we can decode a batch of four items as compared to earlier 2 items. Therefore, we generalize this argument and try to obtain a code that satisfies the parameters we hope to achieve.

Lemma 2 (Gadget lemma [1])**.** *Let C_0 be an (n_0, N_0, k, m) multiset batch code. Then, for any positive integer r there is an (n, N, k, m) multiset batch code $\boldsymbol{C}$ with $n = rn_0$ and $N = rN_0$. We denote the resulting code $\boldsymbol{C}$ by $(r \cdot \boldsymbol{C}_0)$.*

For fixed parameters n, k, we take a parameter l that will allow for the tradeoff between the rate and the number of buckets that can be optimally achieved.

Lemma 3. *For any integers $l \geq 2$ and n, there is a primitive (n, N, k, m) multiset batch code $\boldsymbol{C}_l$ with $n = l$, $N = m = l + 1$, and $k = 2$.*

Proof. We define the encoding function $\mathbf{C}_l$ as $\mathbf{C}_l(x) = (x_1, x_2, \ldots, x_l, (x_1 \oplus x_2 \oplus \ldots \oplus x_l))$. In the above definition we have defined $k = 2$, (i.e) we are able to retrieve any two bit positions of the original database, say $\{i_1, i_2\}$. Therefore we have two conditions:

[3] n is the size of original database

- the bit positions are different $i_1 \neq i_2$
  In this case we can obtain the bits by accessing two different buckets and querying the required bit positions.

- the bit positions are same $i_1 = i_2$
  In this case we can query the bucket having the bit and then we take the XOR of all the other buckets to retrieve the other bit.

□

Lemma 4 (Composition lemma [1]). *Let $\boldsymbol{C}_1$ be an (n_1, N_1, k_1, m_1) batch code and $\boldsymbol{C}_2$ an (n_2, N_2, k_2, m_2) batch code such that the length of each bucket in $\boldsymbol{C}_1$ is n_2 (in particular $N_1 = m_1 n_2$). Then, there is an (n, N, k, m) batch code $\boldsymbol{C}$ with $n = n_1$, $N = m_1 N_2$, $k = k_1 k_2$, and $m = m_1 m_2$. Moreover, if $\boldsymbol{C}_1$ and $\boldsymbol{C}_2$ are multiset batch codes then so is $\boldsymbol{C}$, and if all buckets of $\boldsymbol{C}_2$ have same size then this is also the case for $\boldsymbol{C}$. We will use the notation $\boldsymbol{C}_1 \otimes \boldsymbol{C}_2$ to denote the composed code $\boldsymbol{C}$.*

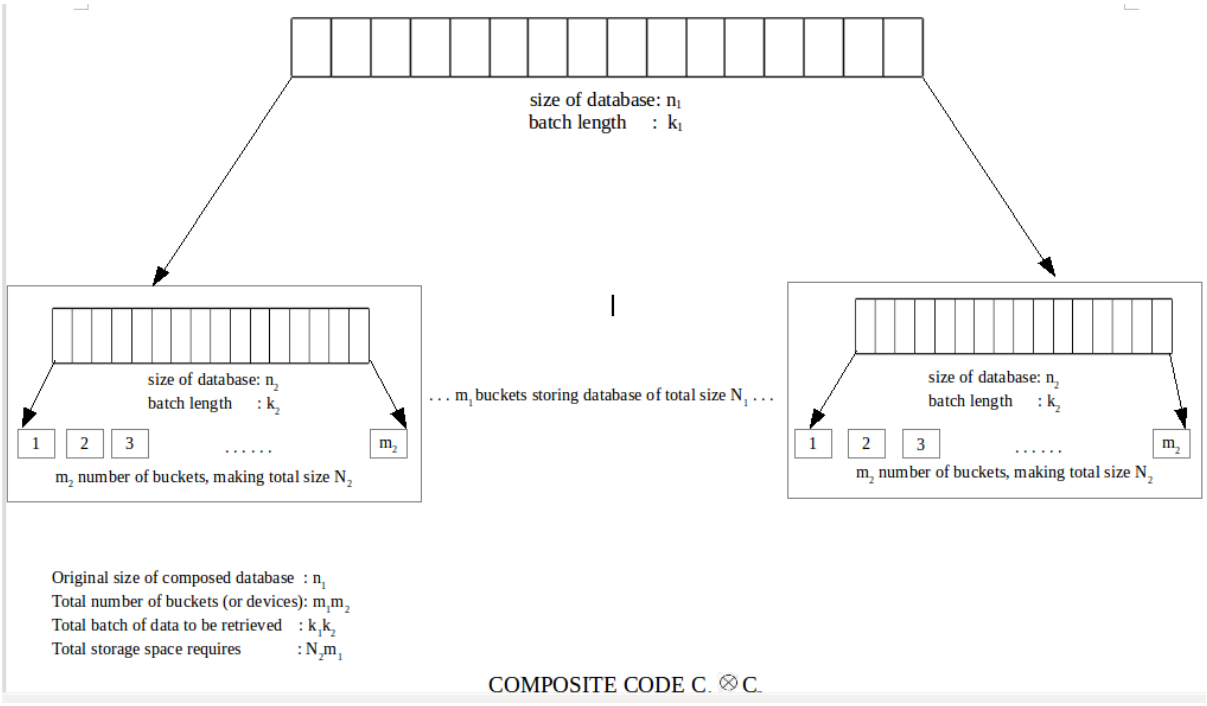

Figure 4: View of composition lemma

The above definition of the composite code of $C_1 \otimes C_2$ can be visualized with the help of Figure 4. It is easy to see that the buckets storing the original database can be subdivided by another batch code, thereby increasing the original length total storage size which the new buckets store (*i.e.* N_2) times the original number of buckets (*i.e.* m_1). Hence, we obtain the following parameters as mentioned in the composition lemma.

Lemma 5. *For integers $l \geq 2$ and $d \geq 1$, there is a (primitive) multiset batch code $\boldsymbol{C}_l^d$ with $n = l^d$, $N = m = (l+1)^d$ and $k = 2^d$.*

Proof. As we can see from Lemma 3 we have a primitive (n, N, k, m) multiset batch code $\mathbf{C}_l$ with $n = l$, $N = m = l+1$, and $k = 2$. We can apply composition of this code d times

thus obtaining the desired code.
More formally we describe $\mathbf{C}_l^d$ inductively as follows $\mathbf{C}_l^1 = \mathbf{C}_l$ and $\mathbf{C}_l^d = (l * \mathbf{C}_l^{d-1}) \otimes \mathbf{C}_l$, where $'*'$ denotes the gadget operator from Lemma 2. □

Theorem 3. *For any integers k, n and $l \geq 2$ there is an explicit multiset batch code with the parameters $m = (l+1)^{\lceil(\log_2 k)\rceil} \approx k^{\log_2(l+1)}$ and $N = \lceil(n/l^d)\rceil m \approx K^{\log_2(1+\frac{1}{l})}n$.*

Proof. From Lemma 5 we can see that there exists a primitive multiset batch code with parameters $n = l^d$ and $k = 2^d$. Let $d = \log_2 k$, then we have $m = (l+1)^{\lceil(log_2 k)\rceil}$ and by lemma 2 we have $N = \lceil(n/l^d)\rceil m$. We know, $x^{\log_2 y} = y^{\log_2 x}$ (take logarithm on both sides). Similarly, $(l+1)^{\lceil\log_2 k\rceil} \approx k^{\log_2(l+1)}$ and $l^{\log_2 k} = k^{\log_2 l}$. Thus, by taking $d = \lceil(\log_2 k)\rceil$ in Lemma 5. We have $m \approx k^{\log_2(l+1)}$ and $N \approx \frac{n}{l^{\log_2 k}} k^{\log_2(l+1)} = k^{\log_2(1+\frac{1}{l})}n$. Here if we set $l = O(\log k)$ we can make the rate of code arbitrarily close to 1. □

Corollary: For any constant $\rho < 1$ and integer k there is an integer $m(= k^{O(\log\log k)})$ such that for all sufficiently large n there is an (n, N, k, m) multiset batch code with $n/N > \rho$.

3.3 Batch Codes from Smooth Codes

Long distance transmission of information can corrupt the data bits and the received data is prone to errors. Therefore, for reliable transmission of information over long range in a noisy channel we make the use of error-correcting codes. These codes are also used in the storage of information over objects that make be susceptible to damage, get corrupted by repeated use or after being stored for a long time. We can try to encode the data into smaller blocks of data items but the problem persists that if a block of data is corrupted that block is completely lost. Also, we can try to encode the whole data set as a single code and store it for further use. But now the difficulty is that even if there is a single error we have to read the whole data set every time. Thus, we need some encoding strategy that can resolve this issue.

This can be achieved in sublinear-time by using the locally decodable code which has the property that we can retrieve an input data item by reading just few data items of the output codeword (possibly corrupted). Formally, locally decodable codes are defined as below.

Definition 8 ((q, δ, ϵ)-locally decodable code)**.** *For fixed δ, ϵ and integer q we say that $\boldsymbol{C}: \{0,1\}^n \to \Sigma^m$ is a (q, δ, ϵ)- locally decodable code if there exists a probabilistic algorithm $\boldsymbol{A}$ such that :*

- *For every $x \in \{0,1\}^n$, for every $y \in \Sigma^m$ with $d(y, \boldsymbol{C}(x)) \leq \delta m$, and for every $i \in [n]$, we have:*

$$Pr[A(y, i) = x_i] \geq 1/2 + \epsilon,$$

  where the probability is taken over the internal coin tosses of $\boldsymbol{A}$

- *In every invocation, $\boldsymbol{A}$ reads at most q indices of y (in fact, without loss of generality, we can assume that $\mathbf{A}$ reads exactly q indices of y).*

An algorithm **A** satisfying the above requirements is called a (q, δ, ϵ)- local decoding algorithm for **C**.

From the locally decodable codes arises the notion of smoothness. In a locally decodable code it may so happen than some bits are queried more than the others, so if these bits get corrupted then we have error in decoding. If we model the querying behaviour such that every bit is probed uniformly at random then it gives rise to the smooth code. Formally, we define smooth code as below:

Definition 9 ((q, c, ϵ)-smooth code)**.** *For a fixed c, ϵ and integer q we say that $\boldsymbol{C} : \{0,1\}^n \to \Sigma^m$ is a (q, c, ϵ)-smooth code if there exists a probabilistic algorithm $\boldsymbol{A}$ such that:*

- *For every $x \in \{0,1\}^n$ and for every $i \in [n]$, we have:*

$$Pr[\boldsymbol{A}(\boldsymbol{C}(x), i) = x_i] \geq 1/2 + \epsilon$$

- *For every $i \in [n]$ and $j \in [m]$, we have:*

$$Pr[\boldsymbol{A}(\cdot, i) \text{ reads index } j] \leq c/m$$

- *In every invocation, $\boldsymbol{A}$ reads at most q indices of y*

(The probabilities are taken over the internal coin tosses of $\boldsymbol{A}$.) An algorithm $\boldsymbol{A}$ satisfying the above requirement is called (q, c, ϵ)- smooth decoding algorithm for $\boldsymbol{C}$.

We say that (q, c, ϵ)- smooth code is also a q-query smooth code when $c = q$ and $\epsilon = 1/2$ (*i.e.*) A always outputs the correct index x_i and reads codewords with probability at most q/m.

We can convert a smooth code into a primitive multiset batch code using the following greedy strategy.

Goal: To successfully decode a batch of k items
Input: The encoded data $\mathbf{C}(x)$, the indices of the items to be recovered $i_1, \ldots, i_k$
Output: $X_1, \ldots, X_k$
Procedure:
while(all items are recovered successfully)
    for $j = 1$ to k
    flag = TRUE
    while(flag)
        invoke the smooth decoder $D(\bar{y})$ to produce a q-tuple of queries
            if (query has been used earlier)
                flag = TRUE
            else
                $x_j = D(\bar{y})$
                flag = FALSE
end.

Thus, the above procedure gives the following theorems:

Theorem 4. *Let $\boldsymbol{C} : \Sigma^n \to \Sigma^m$ be a q-query smooth code. Then $\boldsymbol{C}$ describes a primitive multiset batch code with the parameters (n, m, k, m), where $k = \lfloor m/q^2 \rfloor$.*

Theorem 5. *Let $\boldsymbol{C} : \Sigma^n \to \Sigma^m$ be a q-query smooth code. Then for any k such that $kq/m > \log m$, the code $\boldsymbol{C}$ describes a primitive (n, m, k, m, t) multiset batch code over Σ with $t = kq/m + 2(kq \log m/m)^{1/2}$. Hence for the same t there is also a primitive (n, tm, k, tm) multiset batch code.*

4 Cryptographic Applications

In the previous section we elaborated upon the construction of the batch codes and their use in the general load-balancing situations. Batch codes are also very useful in the Private Information Retrieval (PIR).

Assume a practical situation where data is stored on various servers and we want to retrieve the data bit (say x_i) from a server such that no server is able to know which data bit has been retrieved. The database is modelled as an n bit string $x = x_1 x_2 \dots x_n$, with a computational agent that can do computations based on x and queries made to it. When trying to retrieve data, we want that the servers get absolutely no information whatsoever about the bit. This can be thought of as the situation when we query the a bit position x_i and the database doesn't know this bit but it knows somehow that the i^{th} position is not (let's say) 100, but this conveys some information, which should not be allowed. One naive solution to handle this issue would be to send the complete database to the user querying. But the problem with this solution is that it may not be feasible when the size of the database is quite large (*i.e* it has high communication complexity).

The above setting describes the general PIR protocol. Now we consider the $\binom{n}{k}$-PIR problem, where we are interested in retrieving k-bits from the n-bits string x. A method to achieve this would be by invoking the PIR protocol P k times, independently. The complexity of this would be k times that of P. But we want to obtain signnificant saving in the time complexity as compared to the procedure above. Such amortization can be achieved using the hashing but it has the disadvantage that even if the original PIR scheme is perfectly correct, the amortized scheme is not. Observe that the computational overhead over a single PIR invocation is large.

We can use the batch code to achieve such an amoritzation. Batch codes provides a general reduction from $\binom{n}{k}$-PIR to standard $\binom{n}{1}$-PIR problem. To solve the $\binom{n}{k}$-PIR problem, we fix some (n, N, k, m) batch code to encode the database x. To retrieve the k indices $i_1, i_2, \dots, i_k$, we apply the retrieval procedure to that set; however rather than directly reading one bit from each bucket, the PIR protocol P is applied on each bucket retrieving the bit while maintaining its secrecy. Thus, if $C(n)$ and $T(n)$ be the communication and time complexity of the given PIR-protocol P and $N_1, N_1, \dots, N_m$ be the sizes of buckets, the communication complexity of this protocol is $\sum_{i=1}^{m} C(N_i)$ and its time-complexity is $\sum_{i=1}^{m} T(N_i)$. Thus, this method gives a better reduction while maintaining the secrecy of the bits without errors.

5 Conclusion

In the previous sections we have discussed about the different ways to create a batch code. The batch codes constructed using various procedure satisfies the followig parameters:

Code	rate	m	multiset?
Expander	$1/d < 1/2$	$O(k \cdot (nk)^{(1/(d-1))})$	No
	$\Omega(1/\log n)$	$O(k)$	
Subcube	$\rho < 1$	$k^{O(\log\log k)}$	Yes
Reed Muller	$1/l! - \epsilon < 1/2$	$k \cdot k^{1/(l-1)+o(1)})$	Yes
	$\Omega(1/k^{\epsilon})$	$k \cdot (\log k)^{2+1/\epsilon+o(1)})$	

The dependence of the variables rate and m on the parameters k, n, d can be observed from the table above. Thus, we can use the different parameters according to the required rate and devices m. Also, the batch codes can be used for the amotized PIR protocols thus acheiving a significant saving in time complexity.

References

[1] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai: Batch codes and Their Applications, STOC, pages 262-271. ACM, (2004)

[2] Sergey Yekhanin: Locally Decodable Codes: A Brief Survey. IWCC 2011: pages 273-282

[3] Jonathan Katz, Luca Trevisan: On the Efficiency of Local Decoding Procedures for Error-Correcting Codes, Proceedings of the Thirty-Second Annual ACM Symposium on Theory of computing, 2000, pages 80-86.

[4] Amos Beimel, Yuval Ishai, Eyal Kushilevitz: General constructions for information-theoretic private information retrieval. J. Comput. Syst. Sci. 71(2): 213-247 (2005).

[5] M. Capalbo, O. Reingold, S. Vadhan, and A. Wigderson. Randomness Conductors and Constant-Degree Expansion Beyond the Degree/2 Barrier. In Proc. 34th STOC, pages 659-668, 2002.

[6] A. Ta-Shma, C. Umans, and D. Zuckerman. Loss-less condensers, unbalanced expanders, and extractors. In Proc. 33rd STOC, pages 143-152, 2001.