

МОНГОЛ УЛСЫН ИХ СУРГУУЛЬ  
МЭДЭЭЛЛИЙН ТЕХНОЛОГИ, ЭЛЕКТРОНИКИЙН СУРГУУЛЬ  
МЭДЭЭЛЭЛ, КОМПЬЮТЕРЫН УХААНЫ ТЭНХИМ

Чулуунбаатарын Мөнхбаяр

**Санах ойд байрших түлхүүр-утга өгөгдлийн  
сангийн систем хэрэгжүүлэх**  
**(Implementation of in-memory key-value database system)**

Компьютерын ухаан (D061301)  
Бакалаврын судалгааны ажил

Улаанбаатар

2025 оны 05 сар

МОНГОЛ УЛСЫН ИХ СУРГУУЛЬ  
МЭДЭЭЛЛИЙН ТЕХНОЛОГИ, ЭЛЕКТРОНИКИЙН СУРГУУЛЬ  
МЭДЭЭЛЭЛ, КОМПЬЮТЕРЫН УХААНЫ ТЭНХИМ

Санах ойд байрших түлхүүр-утга өгөгдлийн сангийн систем  
хэрэгжүүлэх  
(Implementation of in-memory key-value database system)

Компьютерын ухаан (D061301)  
Бакалаврын судалгааны ажил

Удирдагч: \_\_\_\_\_ Г.Гантулга  
Гүйцэтгэсэн: \_\_\_\_\_ Ч.Мөнхбаяр (21B1NUM2203)

Улаанбаатар  
2025 оны 05 сар

# Зохиогчийн баталгаа

Миний бие Чулуунбаатарын Мөнхбаяр ”Санах ойд байрших түлхүүр-утга өгөгдлийн сангийн систем хэрэгжүүлэх” сэдэвтэй судалгааны ажлыг гүйцэтгэсэн болохыг зарлаж дараах зүйлсийг баталж байна:

- Ажил нь бүхэлдээ эсвэл ихэнхдээ Монгол Улсын Их Сургуулийн зэрэг горилохоор дэвшүүлсэн болно.
- Энэ ажлын аль нэг хэсгийг эсвэл бүхлээр нь ямар нэг их, дээд сургуулийн зэрэг горилохоор оруулж байгаагүй.
- Бусдын хийсэн ажлаас хуулбарлаагүй, ашигласан бол ишлэл, зүүлт хийсэн.
- Ажлыг би өөрөө (хамтарч) хийсэн ба миний хийсэн ажил, үзүүлсэн дэмжлэгийг дипломын ажилд тодорхой тусгасан.
- Ажилд тусалсан бүх эх сурвалжид талархаж байна.

Гарын үсэг: \_\_\_\_\_

Огноо: \_\_\_\_\_

## ГАРЧИГ

УДИРТГАЛ.....	1
1. СУДАЛГАА .....	2
1.1 Сэдвийн судалгаа .....	2
1.2 Ижил төстэй системийн судалгаа .....	15
1.3 Ашиглагдах технологийн судалгаа.....	20
1.4 Бүлгийн хураангуй .....	29
2. ШИНЖИЛГЭЭ, ЗОХИОМЖ.....	30
2.1 Системийн архитектур .....	30
2.2 Үйл ажиллагааны дараалал .....	30
2.3 Функциональ шаардлага .....	31
2.4 Функциональ бус шаардлага.....	32
3. ХЭРЭГЖҮҮЛЭЛТ .....	33
3.1 Сервер.....	33
3.2 Клиент .....	48
3.3 Үр дүн.....	52
ДҮГНЭЛТ .....	53
НОМ ЗҮЙ .....	53
ХАВСРАЛТ .....	54

## ЗУРГИЙН ЖАГСААЛТ

1.1	HDD соронзон диск .....	2
1.2	SSD соронзон диск .....	3
1.3	Түлхүүр-утга өгөгдлийн сангийн жишээ .....	9
1.4	Системийн жишээ .....	11
1.5	Кейш бүхий системийн жишээ .....	11
1.6	Redis өгөгдлийн бүтэц .....	16
1.7	Memcached ашигласан болон ашиглаагүй үеийн харьцуулалт .....	19
1.8	TCP socket .....	21
1.9	Клиент болон сервер загвар .....	21
1.10	Thread pool загвар .....	22
1.11	Хэш функц жишээ .....	25
1.12	Улаан-хар мод жишээ .....	27
2.1	Системийн архитектур .....	30
2.2	Үйл ажиллагааны дараалал .....	31
3.1	Сервер .....	48
3.2	Клиент .....	52

## ХҮСНЭГТИЙН ЖАГСААЛТ

1.1	NoSQL өгөгдлийн сангийн ангилал .....	8
1.2	Санах ойд суурилсан өгөгдлийн сангуудын жишээ .....	12
1.3	Түлхүүр-утга хосуудыг бичихэд зарцуулсан хугацаа /миллисекунд (мс)/	13
1.4	Түлхүүр-утга хосуудыг бичихэд зарцуулсан санах ойн хэмжээ /мегабайт (мб)/ .....	13
1.5	Түлхүүр-утга хосуудыг бичихэд зарцуулсан хугацаа /миллисекунд (мс)/	13
1.6	Түлхүүр-утга хосуудыг бичихэд зарцуулсан санах ойн хэмжээ /мегабайт (мб)/ .....	14
1.7	түлхүүр-утга хосуудыг бичихэд зарцуулсан хугацаа /миллисекунд (мс)/	14
1.8	Түлхүүр-утга хосуудыг бичихэд зарцуулсан санах ойн хэмжээ /мегабайт (мб)/ .....	14
1.9	Түлхүүр-утга хосуудыг бичихэд зарцуулсан хугацаа /миллисекунд (мс)/	15
1.10	Түлхүүр-утга хосуудыг бичихэд зарцуулсан санах ойн хэмжээ /мегабайт (мб)/ .....	15
1.11	Redis болон Memcached .....	19
1.12	List болон Sets .....	26
3.1	Хугацааны харьцуулалт .....	52

# Кодын жагсаалт

1.1	Thread pool класс . . . . .	22
1.2	ThreadPool класс task гүйцэтгэх функц . . . . .	23
1.3	Thread pool ашигласан кодын жишээ . . . . .	24
3.1	Серверийн хэрэгжүүлэлт . . . . .	35
3.2	Клиентийн хэрэгжүүлэлт . . . . .	49

# УДИРТГАЛ

## Дипломын ажлын зорилго

Санах ой дээр суурилсан түлхүүр-утга өгөгдлийн сангийн серверийг хэрэгжүүлж, олон хэрэглэгчийн хүсэлтийг зэрэг боловсруулж чаддаг, өндөр бүтээмжтэй системийг бүтээх.

## Зорилт

Уг ажлын хүрээнд дараах зорилтуудыг тавьж ажилласан болно.

- Санах ойд суурилсан өгөгдлийн санг уламжлалт диск дээр суурилсан өгөгдлийн сантай харьцуулсан судалгаа хийх;
- Түлхүүр-утга өгөгдлийн сангийн талаар судалгаа хийх;
- Ашиглагдах арга технологийн судалгаа хийх;
- Үр бүтээмжтэй өгөгдлийн бүтцийг ашиглан өгөгдлийн сангийн сервер болон клиент программын хэрэгжүүлэлт хийх.

## Сэдэв сонгосон үндэслэл

Өнөө үеийн мэдээллийн технологийн хөгжилд өгөгдөл боловсруулах хурд, найдвартай байдал, санах ойн үр ашигтай хэрэглээ нь амин чухал хүчин зүйлсийн нэг болж байгаа бөгөөд шуурхай санах ой дээр суурилсан түлхүүр-утга өгөгдлийн сан нь өндөр хурд, бага сааталтай өгөгдөл хадгалах, боловсруулах олон боломжийг бүтээж байна. Энэхүү сэдэв нь онолын мэдлэгийг практикт ашиглах, орчин үеийн олон талт судалгааг судлах, өндөр хурдтай өгөгдлийн сан хөгжүүлэх туршлага хуримтлуулах зэрэг олон давуу талтай тул сонгосон өөрийн дипломын ажлын сэдвээр сонгосон болно.

## Ажлын шинэлэг байдал

Энэхүү судалгааны ажил нь зөвхөн санах ойд суурилсан өгөгдлийн сангаар зогсохгүй түлхүүр-утга өгөгдлийг хэрхэн хурдан, үр ашигтайгаар зохион байгуулах, олон хэрэглэгчийн хүсэлтийг нэгэн зэрэг шийдвэрлэх зэргийг тус бүр нарийвчлан хөгжүүлснээрээ бусад ижил төстэй бакалаврын судалгааны ажлаас онцлогтой юм.



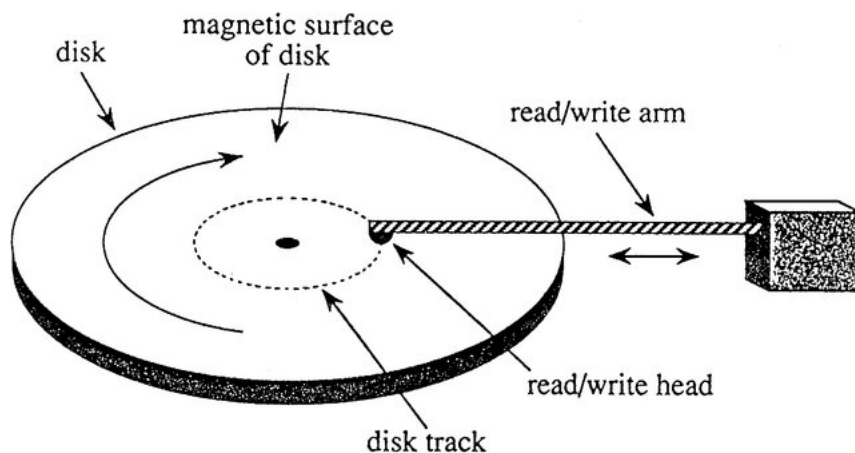
# 1. СУДАЛГАА

Энэ бүлэгт санах ойд суурилсан өгөгдлийн сан болон уламжлалт өгөгдлийн сангийн үзүүлэлтүүд, өргөтгөх чадвар, өгөгдлийг хэрхэн тэсвэртэй хадгалах, ашиглагдах өгөгдлийн бүтэц болон технологиудыг харьцуулах болно. Мөн түүнчлэн Redis технологийн талаар тайлбарлан бусад санах ойд суурилсан өгөгдлийн сангуудын сул болон давуу талуудын талаар судална.

## 1.1 Сэдвийн судалгаа

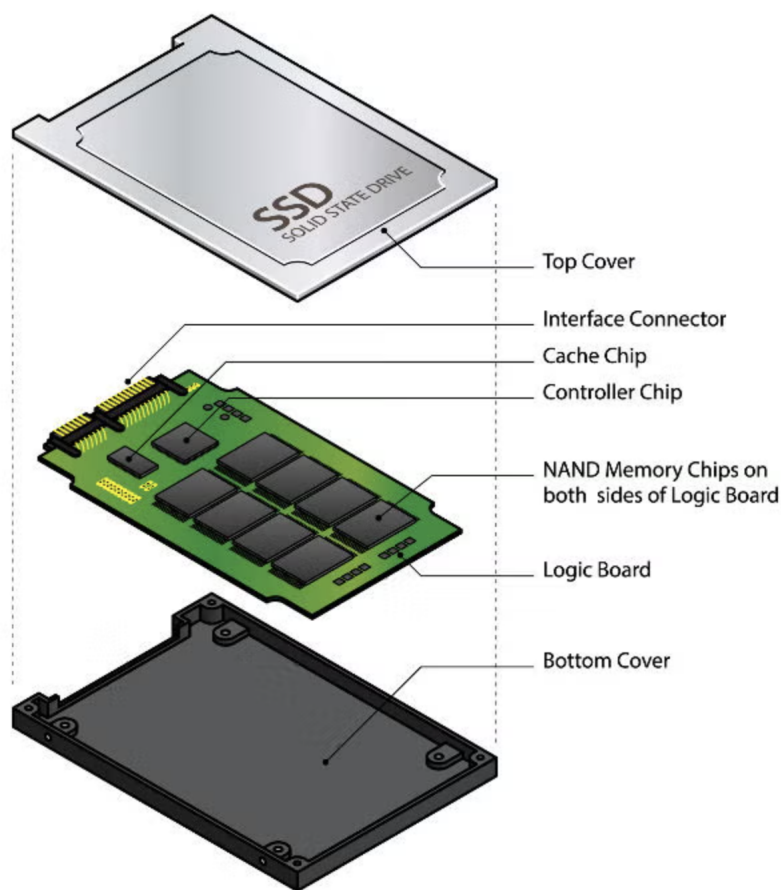
### 1.1.1 Диск (HDDs болон SSDs)

Solid state drives (SSD) болон hard disk drives (HDD) нь дата хадгалах төхөөрөмжүүд юм. HDD үндсэн хэсэг нь platters (диск) хэлбэртэй (Зураг 1.1) ба дата-г цахилгаан цэнэгийн тусламжтайгаар уг диск дээр хадгалдаг. Энэхүү цахилгаан цэнэг нь үйлдэгч гар буюу унших/бичих толгой хэсгээс ирэх ба CPU (Central Processing Unit) зааварчилгааны дагуу ажилладаг. Дискийн тойрог бүр секторт хуваагдах ба сектор бүр жижиг бүрдэл хэсгүүд болох битүүдээс бүрдэнэ. Уг битүүд үйлдэгч гарын үзүүр хэсэгт байх соронзон толгойгоос цахилгаан цэнэгийг хүлээн авч дата-г бичдэг байна.[2] Хүлээн авсан цэнэгүүд бинар луу хөрвөхөд 0-1 секунд зарцуулагддаг.



Зураг 1.1: HDD соронзон диск

SSD нь цахилгаан хэлхээний тусламжтайгаар дата-г блок хэсгүүд дээр хадгалдаг.[3] Зарим SSD дискийн цахилгаан хэлхээ нь тогтворгүй NAND транзисторуудаас бүрдэх NAND (“Not AND” logic gate) флаш санах ой байна. Тогтворгүй NAND транзисторууд нь өгөгдлийг хагас дамжуулагчийн цэнэг болгон цахиур санах ойн чипүүд дээр хадгалдаг. Удирдлага (Controller) хэсгүүд нь бүх флаш санах ойн эсүүдийг удирдах, дата-г хэрхэн тархааж хадгалах зэргийг зохицуулдаг.



Зураг 1.2: SSD соронзон диск

### 1.1.2 HDD болон SSD диск харьцуулалт

- **Багтаамж**

SSD болон HDD дискийн хамгийн том ялгаа нь SSD диск нь соронзон диск бус

флаш санах ой ашигладагт оршино. Дунджаар нэг хэрэглэгч хамгийн ихдээ 8 терабайт(ТБ) хэмжээ бүхий SSD ашиглах боломжтой. 2025 онд ExaDrive 3.5 инч бүхий 32ТБ SSD танилцуулсан ба HDD дискийн хувьд хамгийн ихдээ 20ТБ файл хадгалах боломжтой. Ихэнх зөөврийн болон хувийн компьютерүүд 250 гигабайт(ГБ) хэмжээтэй HDD дисктэй байдаг.

- **Үзүүлэлт**

SSD дискийн үзүүлэлт HDD дискээс хэд дахин өндөр байх ба HDD диск 500 мегабайт/секунд (МБ/С) хурдтай бол ихэнх SSD диск 7000 МБ/С хурдтай.

- **Насжилт**

Насжилтын хувьд HDD нь илүү урт байдаг бол SSD нь бүтэн жилийн турш тэжээлгүй болсны дараа өгөгдөл алдагдах магадлалтай.

### ***1.1.3 Шуурхай санах ой***

Random-access memory (RAM) буюу шуурхай санах ой нь орчин үеийн компьютерын архитектурын үндсэн бүрэлдэхүүн хэсэг бөгөөд өгөгдөл, программыг идэвхтэй боловсруулахад түр хугацаагаар хадгалах үндсэн ажлын санах ой болж өгдөг.[5] Уламжлалт дискийн санах ойн системээс (HDD болон SSD) ялгаатай нь RAM нь зөвхөн тэжээлд залгагдсан үед л өгөгдлийг хадгалдаг, тогтворгүй санах ойн хэлбэр бөгөөд илүү хурдан нэвтрэх хугацаа, санах ойн дурын байршилд жигд хоцрогдолтойгоор өгөгдлийг унших, бичих боломжийг олгодог. Уламжлалт дискийн хадгалалт нь механик бүрэлдэхүүн хэсгүүд болон цуваа хэлбэрийн механизм дээр тулгуурладаг тул миллисекундын хэмжээний хоцрогдол үүсгэдэг. Харин RAM нь наносекундэд хэмжигдэх агшин зуурын хандалтын хугацааг идэвхжүүлдэг хатуу төлөвт электрон хэлхээг ашигладаг. Энэхүү хурдны давуу тал нь RAM-ийг Redis гэх мэт санах ойн түлхүүр-утгыг хадгалалт зэрэг өгөгдөлд байнга хандах, өөрчлөх шаардлагатай программуудад илүү тохиромжтой болгож өгсөн. RAM нь хоёртын мэдээллийг хадгалах чадвартай, сая сая санах ойн эсүүдээс бүрдсэн электрон хэлхээний

нарийн системээр дамжуулан өгөгдлийг хадгалдаг. Эдгээр нүднүүд нь өвөрмөц хаяг бүхий матрицтай төстэй бүтэцтэй болж, завсрын өгөгдлийг дамжих шаардлагагүйгээр санах ойн дурын байршилд шууд нэвтрэх боломжийг олгодог.

#### **1.1.4 TCP/IP**

TCP/IP (Transmission Control Protocol/Internet Protocol) нь интернетэд холбогдсон компьютер системүүдийн хооронд мэдээлэл солилцохыг зохицуулдаг протоколуудын цогц юм. TCP протокол нь холболтод түшиглэсэн, өгөгдлийн дарааллыг хадгалдаг, алдааг шалгадаг, найдвартай байдлыг хангаж өгсөн протокол юм. IP нь пакетуудыг тус тусын зүйлс гэж үздэг бол TCP-г ашиглан дамжуулагч процесс нь өгөгдлөө байтын урсгал болгон дамжуулдаг. Дамжуулагч, хүлээн авагч процессууд нь ижил хурдтайгаар өгөгдөл бичиж, уншиж чадахгүй тул буффер ашиглан хадгалдаг. Хүлээн авагч, дамжуулагч гэсэн 2 буффер байдаг. TCP протокол нь 4 давхаргаас бүрдэнэ.

- Application(Хэрэглээний);
- Transport(Тээврийн);
- Internet(Сүлжээний);
- Link(Өгөгдлийн сувгийн).

#### **APPLICATION LAYER**

Хэрэглээний давхарга нь процесс хооронд мессеж дамжуулах үүрэгтэй ба хэрэглэгчийг үйлчилгээгээр хангадаг. Жишээ нь: цахим шуудан илгээх, файл дамжуулах, веб хуудас гэх мэт.

#### **TRANSPORT LAYER**

Тээврийн давхаргад логик холболт нь end-to-end байдаг. Transport давхаргад нь application давхаргаас сегмент-ийг хүлээн авч үүнийг пакет болгон өөрчлөн network давхаргад хүргэдэг.

#### **INTERNET LAYER**

Сүлжээний давхарга нь интернетийн үндсэн протокол буюу интернет протокол(IP)- ийг агуулдаг. Уг давхаргад пакетын төрлийг датаграм гэж нэрлэдэг. Сүлжээний давхаргын гол үүрэг нь датаграмын үүсвэрээс хүлээн авагч хооронд дамжуулах юм.

#### LINK LAYER

Өгөгдлийн сувгийн давхарга нь фреймийг нэг төхөөрөмжөөс нөгөөд дамжуулах үүрэгтэй. Өөрөөр хэлбэл шууд хоорондоо холбогдсон 2 төхөөрөмжийн хооронд өгөгдлийг дамжуулна.

#### **1.1.5 Өгөгдлийн сан**

Өгөгдлийн сан нь мэдээллийг хадгалах, авах болон удирдах боломжийг олгодог зохион байгуулалт бүхий өгөгдлийн цуглуулга юм.[7] Орчин үеийн өгөгдөлд суурилсан үйл ажиллагаа бүр өгөгдлийн сангийн тусламжтайгаар найдвартай, аюулгүй, хаанаас ч хандах боломжтой болсон ба өгөгдлийн сангийн үндсэн зорилго нь:

1. Өгөгдлийг хадгалах - Өгөгдлийн сан нь их хэмжээний өгөгдлийг уялдаа холбоотой, зохион байгуулалттайгаар хадгалах боломжийг бүрдүүлдэг бөгөөд давхардлыг бууруулж, өгөгдлийн бүрэн бүтэн байдлыг хангах үр ашигтай бүтэц, арга зүйг санал болгодог.
2. Өгөгдлийг олж авах - Хэрэглэгчдэд өгөгдөл дундаас асуулга (Query) ашиглан хайлт хийх боломжийг олгодог нь өгөгдлийг хурдан хугацаанд, үр дүнтэй байдлаар анализ хийх боломжийг олгодог.
3. Өгөгдлийг удирдах - Өгөгдлийн санд хадгалагдаж буй мэдээллийн, найдвартай, уян хатан байдлыг транзакшн(transaction), индекслэлт(indexing), хандах эрхийн удирдлага(access controll) зэрэг механизмуудаар хангаж ажилладаг.

Өгөгдлийн сан нь бүтэц, хэрэглээ, хадгалах арга зэргээс хамааран дараах төрлүүдэд хуваагдана:

- Шаталсан бүтэцтэй (Hierarchical) өгөгдлийн сан;

- Сүлжээ (Network) өгөгдлийн сан;
- Объект хандлага (Object-oriented) өгөгдлийн сан;
- Холбоост (Relational) өгөгдлийн сан;
- Клауд (Cloud) өгөгдлийн сан;
- Төвлөрсөн (Centralized) өгөгдлийн сан;
- Үйлдэл дээр суурилсан (Operational) өгөгдлийн сан;
- NoSQL өгөгдлийн сан.

Дээрх төрлүүдээс Relational (SQL) болон NoSQL нь хамгийн түгээмэл хэрэглэгддэг ба энэхүү судалгааны ажилд холбоо хамааралтай тул цаашид судлах болно.

#### ***1.1.6 Холбоост өгөгдлийн сан***

Relational Database буюу холбоост өгөгдлийн сан нь урьдчилан тодорхойлсон мөр, баганаас бүрдэх ба мэдээллийг асуулга хийн гаргаж авах эсвэл өөрчлөлт хийхийн тулд Structured Query Language (SQL) ашигладаг.[6] Мөн SQL нь ACID (Atomicity, Consistency, Isolation, Durability) шинж чанарыг баталгаажуулснаар, транзакшны бүрэн бүтэн байдал (жишээ нь, санхүүгийн систем, аж ахуйн нэгжийн хүний нөөцийн төлөвлөлт) өндөр байх шаардлагатай программуудад тохиромжтой байдаг. Жишээ нь: MySQL, PostgreSQL, Oracle Database, Microsoft SQL Server гэх мэт.

#### ***1.1.7 NoSQL өгөгдлийн сан***

Динамик схемийг өөрчлөх, өргөтгөх боломжийг олгодог уян хатан байдалд зориулагдсан.

Хагас бүтэцтэй, бүтэцгүй өгөгдөл гэх мэт том хэмжээний, өндөр хурдтай, төрөл бүрийн өгөгдлийн төрлүүдийг боловсруулахад ихэвчлэн ашиглагддаг.

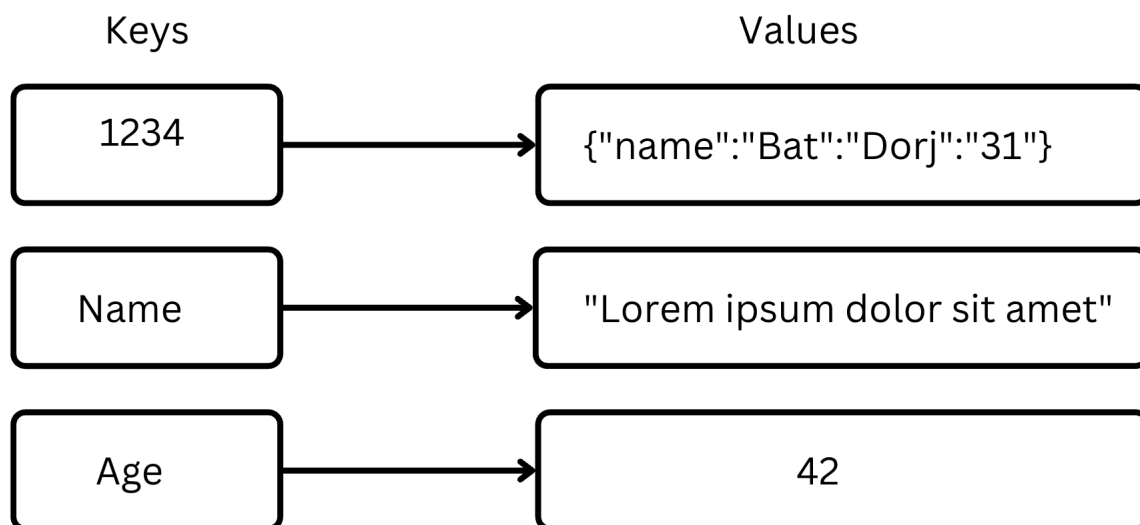
NoSQL өгөгдлийн сангийн төрлүүд:

Төрөл	Ажиллах зарчим	Жишээ
Баримт бичиг	JSON-тэй төстэй уян хатан баримт бичигт өгөгдлийг хадгалах	MongoDB, CouchDB
Багана - Гэр бүлээр (Column-Family)	Хүснэгтийн өгөгдлийг хуваалцах, өргөтгөх боломжтой хадгалах зориулалттай	Apache Cassandra, HBase
Граф (Graph)	Сүлжээнд холбогдсон өгөгдлийн нарийн төвөгтэй асуулгад тохиромжтой байдлаар хадгална	Neo4j, ArangoDB
Түлхүүр-утга (Key-value)	Өгөгдлийг түлхүүр-утгын хос хэлбэрээр хадгалах энгийн бүтэц	Redis, Amazon DynamoDB

Table 1.1: NoSQL өгөгдлийн сангийн ангилал

### 1.1.8 Түлхүүр-утга өгөгдлийн сан

Түлхүүр-утга өгөгдлийн сан нь зөвхөн түлхүүр түүнд харгалзах утгыг хадгалдаг NoSQL өгөгдлийн сангийн нэг төрөл юм. Бусад өгөгдлийн сантай харьцуулахад энгийн бөгөөд нарийн төвөгтэй бус бүтэцтэй тул хамгийн хурдан өгөгдлийн сан юм. Тодорхой нэг утгыг (энэ нь энгийн тоо, тэмдэгт мөрөөс эхлээд нарийн бүтэцтэй объект хүртэл байж болно) цорын ганц давхцахгүй түлхүүртэй(ихэвчлэн тоо, тэмдэгт мөр) холбож хадгалдаг.[1] Харин өгөгдлийг олж авахдаа тухайн түлхүүрийг ашиглан түүнд харгалзах утгыг дуудаж авдаг.



Зураг 1.3: Түлхүүр-утга өгөгдлийн сангийн жишээ

### 1.1.9 Санах ойд суурилсан түлхүүр-утга өгөгдлийн сан

Санах ойд суурилсан түлхүүр-утга өгөгдлийн сан нь өндөр хурдаар өгөгдлийг RAM дээр хадгалдаг. MySQL, PostgreSQL зэрэг диск дээр суурилсан уламжлалт өгөгдлийн сангууд нь өгөгдлийг тэсвэртэй байдлаар хадгалдаг боловч диск дээр бичих, унших хурд удаан байдаг. Харин RAM дээр суурилсан түлхүүр-утга өгөгдлийн сан нь уламжлалт түлхүүр-утга сангуудтай төстэй боловч өгөгдлийг санах ойд хадгалснаар илүү өндөр гүйцэтгэл үзүүлдэг ба өгөгдлийг маш хурдан унших, бичих боломжийг олгодог тул өндөр хурд шаардлагатай системүүдэд ашиглагддаг. Санах ойд суурилсан өгөгдлийн сан нь өндөр хурдаар өгөгдлийг олж авах боломж олгодог хэдий ч тухайн төхөөрөмж унтарч асах үед устаж алга болно. Энэ асуудлыг снапшот (Snapshot) болон Image ашиглан шийдсэн. Гэхдээ энэ нь арай их зардал гарах шаардлагатай болдог тийм учраас ихэнх компаниуд зөвхөн ашиглалтын хурд өндөр байх шаардлагатай нөхцөлд л санах ойд суурилсан өгөгдлийн санг ашигладаг. 21-р зуунд хамгийн их ашиглагдаж буй санах ойд суурилсан өгөгдлийн сангуудын хамгийн том төлөөлөгч бол кэйш технологи байсан ба веб хөтөч болон веб холболт (Session) кэйш

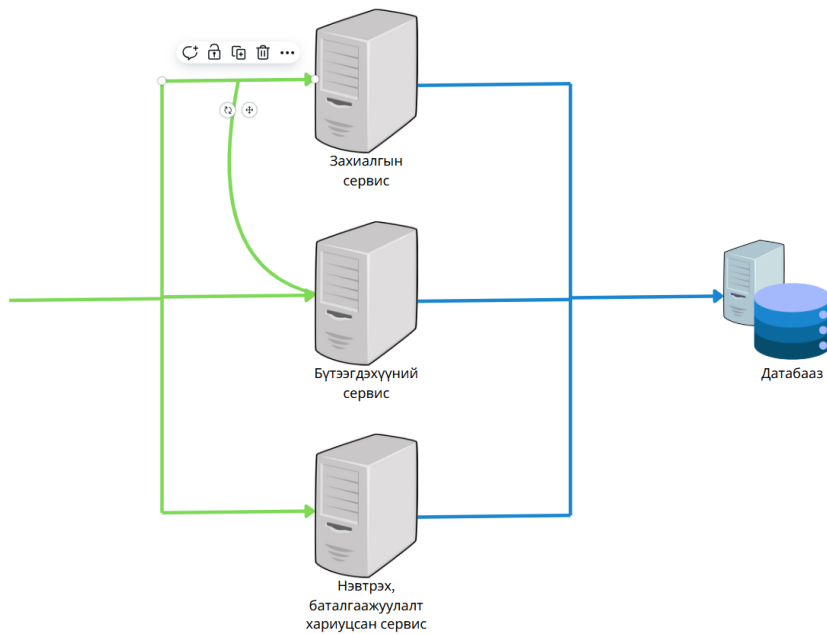


хийснээр ихэнх вебсайтууд backend өгөгдлийн сангаас өгөгдлийг илүү хурдан уншиж амар болсон.

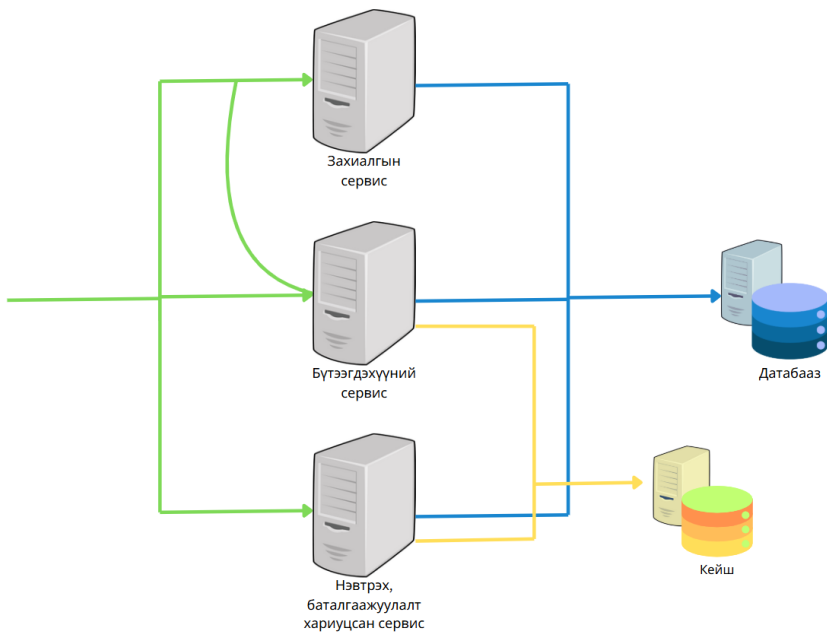
#### **1.1.10 Кейш технологи**

Redis болон Memcached хамгийн өргөн тархсан нээлттэй эхийн технологиуд болно. Кейш технологи гарч ирэхээс өмнө компаниуд их хэмжээний өгөгдлийг уншиж бичих асуудалтай тулгарсан ба энэ асуудлын шийдэл нь зөвхөн унших зориулалттай датаг зохицуулалт байв.[4] Хэрэглэгчийн уншиж, бичиж буй бүх үйлдлүүдийг өгөгдлийн сангаас унших нь цаг хугацааны хувьд зарцуулалт өндөр, тухайлбал хэрэглэгчийн нэвтрэлт баталгаажуулалт, захиалга, бүтээгдэхүүн гэсэн 3 систем бүхий (Зураг 1.4) байгууллагын жишээг үзүүлсэн ба захиалгын системээс бусад нь харьцангуй тогтвортой буюу өөрчлөлт арай удаан орох юм. Тиймд энэ 2 систем дээр кэйш технологи ашиглан (Зураг 1.5) өгөгдлийн сангаас унших үйлдлийг багасгасан байна. Ингэснээр бичилт хийх зай илүү их үлдэж, системийн унших үзүүлэлтийг хурдасгасан юм. Үүний дараа тухайн кэйш-д хадгалагдаж байгаа өгөгдөл үндсэн өгөгдлийн сан дээр өөрчлөлт орсон бол хэрхэн өөрчлөлтийг цаг тухайд нь авах асуудал тулгарсан ба үүнд дараах шийдлүүд байдаг.

- TTL (Time to live ) утгууд нь кэйш хийх тодорхой хугацаа тавина;
- Batch updates - өгөгдлийн санд өөрчлөлт орсон эсэхийг шалгах процесс үе үе шалгаад кэйшийг шинэчлэн;
- Dynamic update - Өгөгдөл нэмэх, устгах, өөрчлөх үед кэйшийг update хийнэ.



Зураг 1.4: Системийн жишээ



Зураг 1.5: Кейш бүхий системийн жишээ

### 1.1.11 Үзүүлэлт

2017 онд Кинг Сауд их сургуулийн хийсэн судалгаанд санах ойд суурилсан өгөгдлийн сангуудын үзүүлэлтийг хугацаа ба санах ой зарцуулалтаар нь Ubuntu 14.04 (64-bit) үйлдлийн систем, 16 GB санах ой, CPU Intel Core i7-4710MQ , ext4 файл систем, Java version 1.8.060 бүхий туршилтын орчинд хийж гүйцэтгэсэн. Үүнд:

- Нэг түлхүүр-утгын хосыг бичих үзүүлэлт;
- Нэг түлхүүрт харгалзах утгыг унших үзүүлэлт;
- Нэг түлхүүрт харгалзах түлхүүр-утгыг устгах үзүүлэлт;
- Өгөгдлийн санд байгаа бүх өгөгдлийг авах үзүүлэлт.

Туршилт хийсэн технологиуд:

Доорх туршилтад ашиглагдсан түлхүүр-утгуудыг санамсаргүй байдлаар гаргасан.

Өгөгдлийн сангийн нэр	Өгөгдлийн сангийн загвар	Хувилбар
MongoDB	Баримт бичиг	3.2.4
Redis	Түлхүүр-утга	3.0.7
Memcached	Түлхүүр-утга	1.4.14
Cassandra	Багана	2.2.5

Table 1.2: Санах ойд суурилсан өгөгдлийн сангуудын жишээ

## Нэг түлхүүр-утгын хосыг бичих үзүүлэлт

Table 1.3: Түлхүүр-утга хосуудыг бичихэд зарцуулсан хугацаа /миллисекунд (мс)/

Өгөгдлийн сан	Бичилтийн тоо			
	1,000	10,000	100,000	1,000,000
Redis	34	214	1666	14.638
MongoDB	904	3482	26.030	253.898
Memcached	23	100	276	2813
Cassandra	1202	4487	15.482	140.842

Table 1.4: Түлхүүр-утга хосуудыг бичихэд зарцуулсан санах ойн хэмжээ /мегабайт (мб)/

Өгөгдлийн сан	Бичилтийн тоо			
	1,000	10,000	100,000	1,000,000
Redis	2.5	3.8	4.3	62.7
MongoDB	56.9	263.6	365	155.9
Memcached	5.3	27.2	211	264.9
Cassandra	5.3	7.5	208.1	102.6

## Нэг түлхүүрт харгалзах утгыг унших үзүүлэлт

Table 1.5: Түлхүүр-утга хосуудыг бичихэд зарцуулсан хугацаа /миллисекунд (мс)/

Өгөгдлийн сан	Бичилтийн тоо			
	1,000	10,000	100,000	1,000,000
Redis	8	6	8	8
MongoDB	8	10	11	13
Memcached	9	14	14	30
Cassandra	2	2	3	6

Table 1.6: Түлхүүр-утга хосуудыг бичихэд зарцуулсан санах ойн хэмжээ /мегабайт (мб)/

Өгөгдлийн сан	Бичилтийн тоо			
	1,000	10,000	100,000	1,000,000
Redis	1.3	1.3	1.3	1.3
MongoDB	1.3	2.5	2.5	1.3
Memcached	1.3	2.5	1.3	2.5
Cassandra	0	0	0	0

### Нэг түлхүүрт харгалзах түлхүүр-утгыг устгах үзүүлэлт

Table 1.7: түлхүүр-утга хосуудыг бичихэд зарцуулсан хугацаа /миллисекунд (мс)/

Өгөгдлийн сан	Бичилтийн тоо			
	1,000	10,000	100,000	1,000,000
Redis	0	1	0	0
MongoDB	75	88	92	355
Memcached	17	17	16	13
Cassandra	2	2	1	2

Table 1.8: Түлхүүр-утга хосуудыг бичихэд зарцуулсан санах ойн хэмжээ /мегабайт (мб)/

Өгөгдлийн сан	Бичилтийн тоо			
	1,000	10,000	100,000	1,000,000
Redis	0	0	0	0
MongoDB	10	10	10	11.3
Memcached	2.2	2.1	2.2	2.2
Cassandra	0	0	0	0

### Өгөгдлийн санд байгаа бүх өгөгдлийг авах үзүүлэлт

Table 1.9: Түлхүүр-утга хосуудыг бичихэд зарцуулсан хугацаа /миллисекунд (мс)/

Өгөгдлийн сан	Бичилтийн тоо			
	1,000	10,000	100,000	1,000,000
Redis	10	9	11	11
MongoDB	9	15	9	8
Cassandra	9	32	24	54

Table 1.10: Түлхүүр-утга хосуудыг бичихэд зарцуулсан санах ойн хэмжээ /мегабайт (мб)/

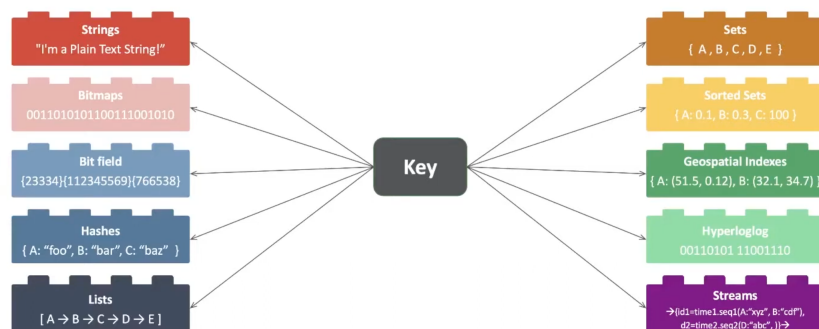
Өгөгдлийн сан	Бичилтийн тоо			
	1,000	10,000	100,000	1,000,000
Redis	2.1	2.1	2.2	2.2
MongoDB	1.3	1.3	1.1	0.8
Cassandra	0.6	0.6	1.3	1.3

## 1.2 Ижил төстэй системийн судалгаа

### 1.2.1 Redis

Redis (Remote Dictionary Server) нь 2009 онд Сальваторе Санфилипо гэх Италийн программ хөгжүүлэгчийн бүтээсэн санах ойд суурилсан өгөгдлийн бүтэц хадгалах систем бөгөөд голчлон өгөгдлийн сан, кэш болон мессеж брокерын (message broker) үүргийг гүйцэтгэдэг. Redis нь тэмдэгт мөр (strings), жагсаалт(list), багц(set), эрэмбэлэгдсэн багц (sorted set), хэш (hash), битмап (bitmap), газарзүйн индекс (geospatial index) гэх мэт олон төрлийн өгөгдлийн бүтцүүдийг дэмждэг.

SSD/HDD дээр суурилсан өгөгдлийн сантай харьцуулахад санах ой дээр ажилладаг тул унших, бичих үйлдлүүд нь агшин зуур хийгддэг. Санах ойд хадгалдаг хэдий ч Redis нь сервер унах эсвэл дахин ачаалахад өгөгдөл алдагдахаас зайлсхийхийн тулд өгөгдлийг дискэнд хадгалах хоёр үндсэн (RDB(Redis DataBase) ба AOF(Append-Only File)) механизмтайгаараа бусад ижил төстэй системүүдээс ялгарсан. RDB нь Redis-ийн оператив санах ойн төлөвийн үечилсэн зургийг үүсгэж, хоёртын форматтайгаар дискэнд хадгалах механизм юм харин



Зураг 1.6: Redis өгөгдлийн бүтэц

AOF нь Redis-д өгөгдлийг өөрчилдөг бүх командуудыг лог файл (append-only file) -д бичих механизм юм. SET, INCR, DEL гэх мэт үйлдлүүд файлын төгсгөлд нэмэгддэг.

## Redis ашигладаг хэрэглэгчийн жишээ

- **Кэйшлэлт (Caching)**

- Веб кэйшлэлт: Хэрэгцээ ихтэй хайлтын үр дүнг хадгалах (HTML хуудасны хэсэг, өгөгдлийн сангийн үр дүн гэх мэт).
- Session кэйш: Хэрэглэгчийн нэвтрэлт, тохиргоо, төлөв зэрэг мэдээллийг хурдан сэргээх.

- **Мэдээлэл дамжуулалт (Message Broker / Pub/Sub)**

- Бодит агшны мэдэгдэл: Чат мессеж, статусын өөрчлөлт гэх мэт.
- Event-driven архитектур: Микро сервисүүдийн хооронд мэдээлэл дамжуулах.

- **Бодит агшны анализ (Real-Time Analytics)**

- Бодит агшны тооцоолол: Вэбсайт шууд үзэлт, API дуудах, хэрэглэгчийн үйлдлийг тоолох.
- Эрэмбэлэгдсэн жагсаалт: Хэрэглэгчийн оноо, өрсөлдөөнт платформд ашиглах.

- **Өгөгдлийн бүтэц хадгалах (Data Structure Storage)**

- Дараалал (Queue): Таскуудын урсгалыг удирдах.
- Hash: Хэрэглэгчийн профайл гэх мэт өгөгдлийг зохион байгуулах.

- **Газарзүйн өгөгдөл (Geospatial Indexing)**

- GPS-ийн өгөгдөл хадгалах.
- Ойролцоох объект хайх: Дэлгүүр, ресторан олох гэх мэт.

- **Distributed Locking**

- Олон процесс нэгэн зэрэг өгөгдөлд хандахаас сэргийлэх.

- **Өгөгдөл хадгалах, нөөцлөх (Replication & Persistence)**

- Redis-д хадгалагдаж буй өгөгдлийг олон серверт хуулбарлах, хадгалах боломжтой.

- **Job Scheduling**

- Redis-ийг сервер дээрх job- уудыг хуваарилах байдлаар ажиллуулж болно.

## **Redis технологи ашиглаж буй компаниуд**

### **Netflix: Кино санал болгох кэйшлэлт**

Netflix хэрэглэгчдийн үзсэн кино, сонирхолд тулгуурлан шинэ кино санал болгохын тулд Redis ашиглаж, өгөгдлийг хурдан боловсруулах, хадгалах кэш болгон хэрэглэдэг.

### **Twitter: Цаг хугацааны дарааллыг зохицуулах**

Twitter нь хэрэглэгчдийн жиргээг цаг хугацааны дарааллаар харуулахын тулд Redis болон EvCaching -д өгөгдлийг хурдан хадгалж, хялбархан авах боломжтой болгож өгдөг.



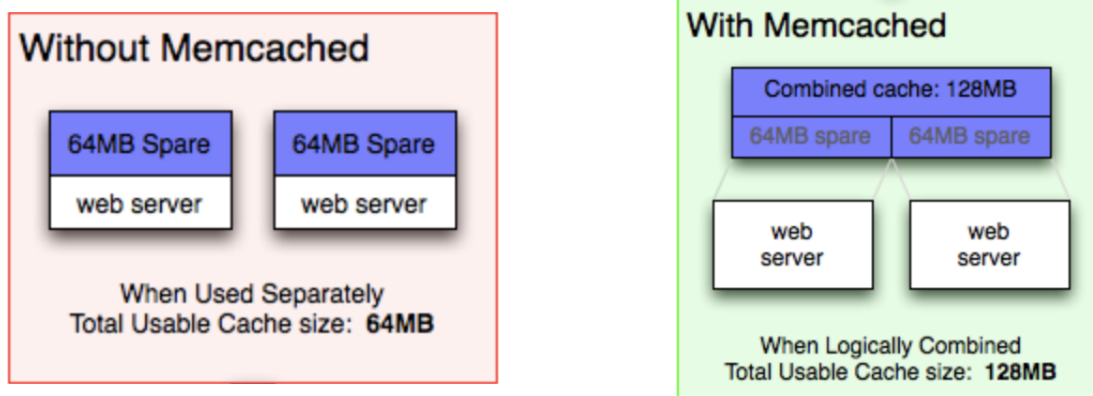
## **GitHub: Background job боловсруулах**

GitHub нь кодын өөрчлөлт, мэдэгдэл илгээх, хайлт хийх зэрэг background job-г хялбар, хурдан зохицуулахын тулд Redis ашигладаг.

### **1.2.2 Memcached**

Memcached-ийг Брэд Фицпатрик 2003 оны 5-р сарын 22-нд LiveJournal веб сайтдаа зориулан анх Perl хэл дээр хөгжүүлсэн боловч дараа нь C хэл дээр Анатоли Воробей дахин бичжээ. Memcached нь өндөр гүйцэтгэлтэй, тархмал санах ойд суурилсан объект кэйшлэх систем бөгөөд анхны зорилго нь динамик веб програмуудын гүйцэтгэлийг сайжруулахын тулд өгөгдлийн сангийн ачааллыг багасгах зорилгоор бүтээгдсэн. Memcached нь санах ойг илүү үр ашигтай ашиглах боломжийг олгоно. Доорх диаграммыг харвал хоёр төрлийн байршуулалтын хувилбарыг харуулсан байна:

1. Бүх серверүүд бие даасан байдлаар ажиллана - Хоёр серверийн кэйшийн хүчин чадал тус бүр 64MB. Хэрэв тус тусдаа ажиллавал нийт ашиглах боломжтой кэйшний хэмжээ 64MB байна.
2. Нэг сервер бусад серверүүдийн санах ойг ашиглах боломжтой - Memcached ашигласнаар, серверүүдийн хоорондын кэйшний хүч чадал нэгтгэгдэж, нийтдээ 128MB кэш болж байна.



Зураг 1.7: Memcached ашигласан болон ашиглаагүй үеийн харьцуулалт

### 1.2.3 Redis болон Memcached

Table 1.11: Redis болон Memcached

Онцлог	Memcached	Redis
Өгөгдлийн бүтэц	Түлхүүр-утга (string)	Түлхүүр-утга (string, List, Set, Hash, Sorted Set, Bitmap, HyperLogLog, Stream)
Хэмжээ	Хамгийн ихдээ 1MB өгөгдөл хадгалах	1GB хүртэлх өгөгдөл хадгалах (хэмжээ нь өөрчлөгдөж болно)
Хадгалалт	Өгөгдлөө Зөвхөн санах ойд хадгална	RDB, AOF ашиглан өгөгдлийг дискэнд хадгалж болно
Threading	Multi-threaded	Single-threaded

## 1.3 Ашиглагдах технологийн судалгаа

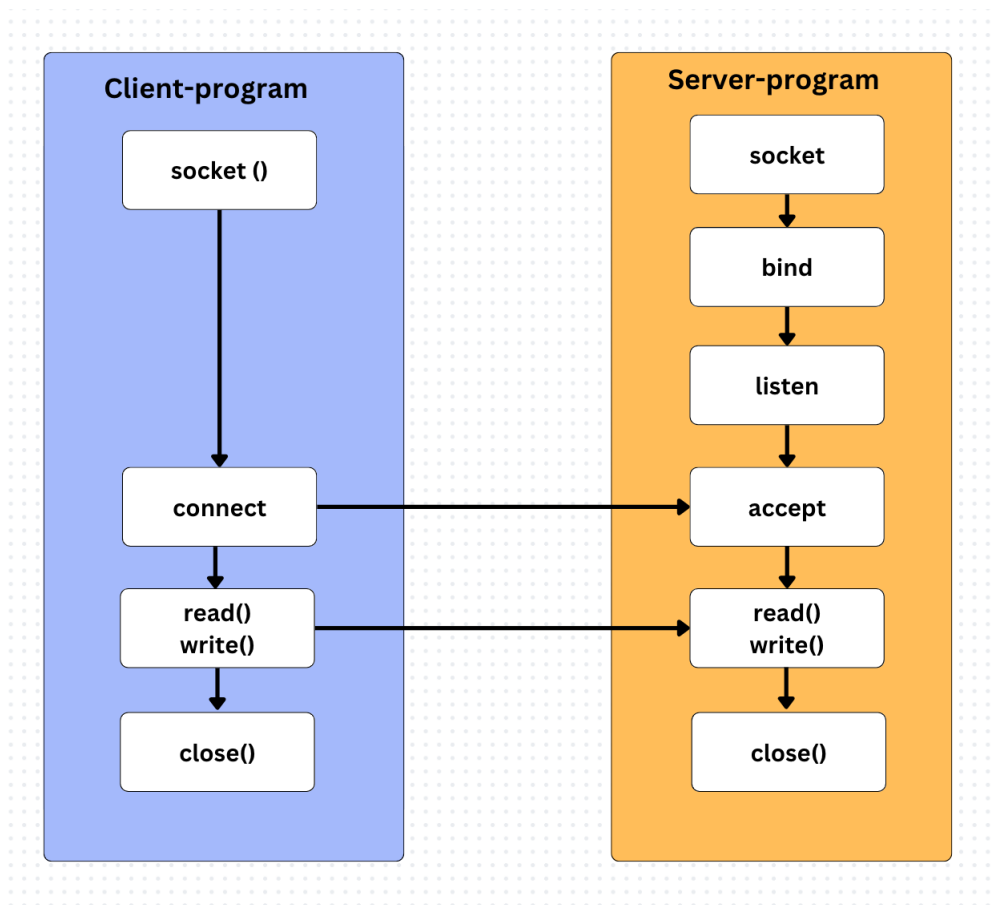
### 1.3.1 C++

C++ нь Standard Template Library(STL)- р дамжуулан өндөр үр ашигтай өгөгдлийн бүтцийг ашиглах боломж олгодог тул санах ойд суурилсан өгөгдлийн санд хамгийн тохиромжтой. Жишээлбэл, `std::unorderedmap` (хэш хүснэгт) болон `std::map` (тэнцвэржүүлсэн мод) зэрэг бүтэц нь хайх, оруулах, устгах үйлдлийг хурдан гүйцэтгэдэг тул түлхүүр-утгын хадгалалтыг хэрэгжүүлэхэд түгээмэл хэрэглэгддэг. Эдгээр өгөгдлийн бүтэц нь маш оновчтой бөгөөд их хэмжээний өгөгдөлтэй ажиллах эсвэл зэрэгцээ хандалтыг дэмжих гэх мэт энэхүү судалгааны ажилд тохиромжтой тул сонгон авсан болно.

### 1.3.2 TCP socket

Сокет гэдэг нь сүлжээгээр хоёр төхөөрөмж хооронд өгөгдөл солилцох интерфейс юм. Сокет нь сүлжээний API (application programming interface)-ийн нэг хэсэг бөгөөд өгөгдлийг илгээх, хүлээн авах зориулалттай функцуудыг агуулдаг.Сокетийг ашигласнаар програм хангамжийн түвшинд сүлжээний холболт үүсгэж, өгөгдлийг уян хатан удирдах боломжтой болдог. TCP нь найдвартай, холболтод суурилсан протокол бөгөөд өгөгдлийг алдалгүй, зөв дарааллаар хүргэдэг. TCP сокет нь TCP протоколыг ашиглан хоёр програмын хооронд найдвартай холболт үүсгэж, түүгээр дамжуулан мэдээлэл солилцдог механизм юм. Сокет нь ихэнхдээ клиент-сервер програмуудад хэрэглэгддэг. Сервер болон клиент нь (Зураг 1.8) дээрх логикуудаас бүрдэнэ.

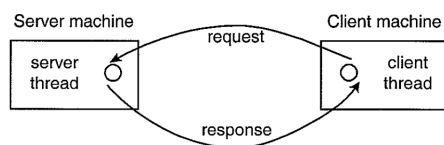
- Сервер тал сокет үүсгэж, үүнийг тодорхой портод холбодог;
- Дараа нь хүлээлгийн төлөвт орж, клиент ирэхийг хүлээнэ;
- Клиент мөн өөрийн сокетийг үүсгэж, сервер рүү холбогдох оролдлого хийнэ;
- Холболт амжилттай болсны дараа өгөгдөл солилцоо явагддаг.



Зураг 1.8: TCP socket

### 1.3.3 Олон урсгалт сан (Multi-thread pool)

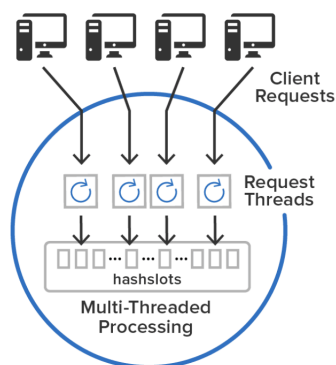
Клиент болон сервер загвар нь хамгийн өргөн хэрэглэгддэг програмчлалын загваруудын нэг юм (Зураг 1.9). Энэ загварт үйлчлүүлэгчид шаардлагатай үед үйлчлүүлэгчийн машинаас



Зураг 1.9: Клиент болон сервер загвар

сервер рүү хүсэлт илгээдэг бөгөөд үүнд серверийн процесс нь тодорхой порт дээр сонсох

замаар ирж буй хүсэлтийг хүлээж байдаг. Өөрөөр нэг дор нэг хүсэлтийг шинэ урсгал (цаашид thread гэх) үүсгээд, хариуг нь буцааж ажилладаг гэвч олон хэрэглэгч нэг дор хандах үед энэ загвар нь гацалтад орж ажиллагаа нь удааширна. Энэ асуудлын шийдэл нь олон урсгалт сан бөгөөд загварын хувьд урьдчилан үүсгэгдсэн ажилчин урсгалууд (worker threads) болон шинэ таскуудын дарааллаас бүрдэнэ (Зураг 1.10). Thread бүр idle эсвэл workifig гэсэн хоёр



Зураг 1.10: Thread pool загвар

төлөвийн аль нэгт байх ба шинэ task ачааллах үед idle thread дарааллаас гарч worker thread дараалалд орно. Task дууссаны дараа буцаад idle thread дараалалд орж дараагийн үйлдлийг хүлээнэ. Энэхүү шийдэл нь уламжлалт шийдлээр арав дахин хурдан юм. Thread pool -н C++ хэл дээр хэрэгжүүлэлтийн жишээ (Код 1.1): ThreadPool класс нь public төрлийн байгуулагч функц, устгагч функц болон таскуудын дараалал, private төрлийн вектор (thread-н цувааг хадгална), mutex (дундын өгөгдлийг давхар хандахаас сэргийлэх класс) зэргээс бүрдэнэ.

```

1  ...
2  #include <thread>
3  class ThreadPool {
4  public:
5      ThreadPool(size_t numThreads);
6      ~ThreadPool();
7      void enqueue(std::function<void()> task);
8  
```

```

9 private:
10     std::vector<std::thread> workers;
11     std::queue<std::function<void()>> tasks;
12     std::mutex queueMutex;
13     std::condition_variable condition;
14     bool stop = false;
15 };

```

Код 1.1: Thread pool класс

```

1 ThreadPool::ThreadPool(size_t numThreads) {
2     for (size_t i = 0; i < numThreads; ++i) {
3         workers.emplace_back([this] {
4             while (true) {
5                 std::function<void()> task;
6                 {
7                     std::unique_lock<std::mutex> lock(queueMutex);
8                     condition.wait(lock, [this] { return stop || !tasks
9                         .empty(); });
10                    if (stop && tasks.empty()) return;
11                    task = std::move(tasks.front());
12                    tasks.pop();
13                }
14                task();
15            }
16        });
17    }

```

Код 1.2: ThreadPool класс task гүйцэтгэх функц

Thread-н тоог статикаар зааж өгнө. Доорх жишээн дээр 4 thread дээр 10 task ажиллуулсан байна. (Код 1.3)

```
1 int main() {  
2     ThreadPool pool(4);  
3     for (int i = 0; i < 10; ++i) {  
4         pool.enqueue([i] { std::cout << "Task " << i << " executed\n";  
5             });  
6     }  
7     std::this_thread::sleep_for(std::chrono::seconds(1));  
8 }
```

Код 1.3: Thread pool ашигласан кодын жишээ

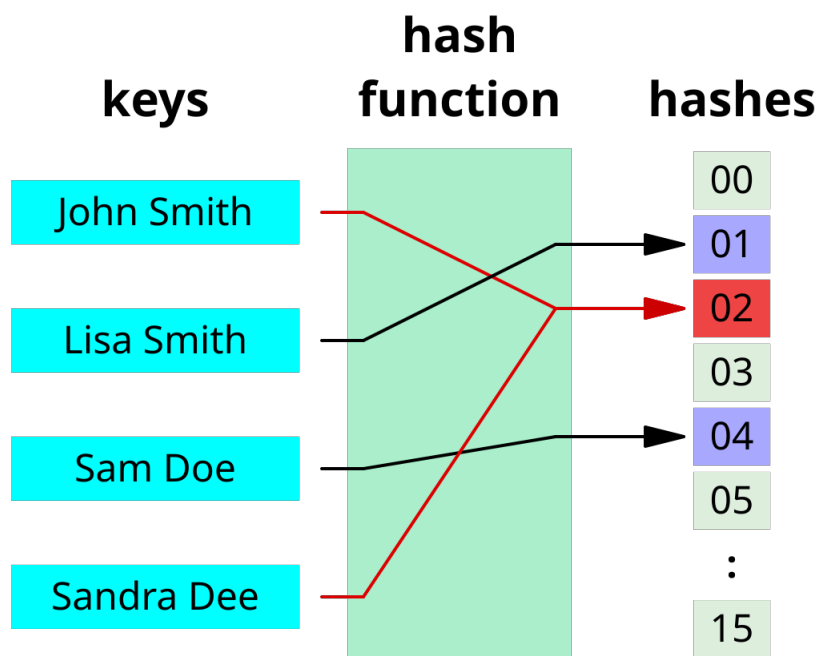
### Тэмдэгт мөр (String)

Тэмдэгт мөр нь компьютерын шинжлэх ухаанд өргөн хэрэглэгддэг өгөгдлийн төрөл бөгөөд тэмдэгтүүдийн дарааллыг илэрхийлдэг. Энэ нь текстэн мэдээлэл хадгалах, боловсруулах, хайх, өөрчлөхөд чухал үүрэг гүйцэтгэдэг. Програмчлалын хэлүүдэд тэмдэгт мөрийг олон янзаар хэрэгжүүлдэг бөгөөд өгөгдлийн сан, хайлтын систем, кэш санах ой зэрэгт өргөн ашигладаг.

### Хэш (Hash) хүснэгт

Орчин үеийн компьютерын шинжлэх ухаан, мэдээллийн сан болон өгөгдөл боловсруулах технологид хэш хүснэгт нь хамгийн өргөн хэрэглэгддэг өгөгдлийн бүтэц юм. Энэ нь түлхүүр-утга (key-value) гэсэн хос мэдээллийг хадгалж, хайх, нэмэх, устгах үйлдлийг маш хурдан гүйцэтгэх боломжийг олгодог. Хэш хүснэгт нь өгөгдлийг хадгалахдаа хэш функц (hash function) ашиглан түлхүүрийг тоон утга болгон хувиргаж, массив хэлбэрийн хүснэгтэд хадгалдаг. Ингэснээр өгөгдлийг хайх, нэмэх, устгах үйлдлийг маш хурдан (дунджаар  $O(1)$  хугацаанд)

хийх боломжтой. Хоёр өөр түлхүүр ижил хэш индекс рүү оноогдох тохиолдлыг мөргөлдөөн (collision) гэнэ. Мөргөлдөөн үүсвэл нэмэлт Chaining (Гинжилсэн жагсаалт), Open Addressing (Нээлттэй хаяглалт) алгоритмууд ашиглана.



Зураг 1.11: Хэш функц жишээ

Хэш функц нь түлхүүрийг тоон индекс болгон хувиргах математик функц юм. Хэш функцийн үндсэн зорилго нь өгөгдлийг жигд тархах бөгөөд мөргөлдөөн (collision)-ийг багасгах явдал юм. Хэш функц нь дараах төрлүүдтэй:

1. Modulo Hashing – Хамгийн энгийн арга бөгөөд түлхүүрийг хүснэгтийн хэмжээгээр хуваах замаар индексийг тодорхойлдог.
2. Multiplication Hashing – Хурдан бөгөөд тогтвортой хэш утга гаргахын тулд тогтмол утга (A) ашигладаг.
3. Cryptographic Hashing (SHA, MD5 гэх мэт) – Өгөгдлийг нууцлалтай хадгалах, аюулгүй байдлыг хангах үед ашиглагддаг.



#### 4. Universal Hashing – Санамсаргүй хэш функцүүдийг ашиглаж, мөргөлдөөнийг багасгах арга.

### Дараалал (Queue)

Queue (дараалал) нь FIFO (First In, First Out) зарчим дээр суурилсан өгөгдлийн бүтэц бөгөөд олон төрлийн алгоритм, системийн ажиллагаанд өргөн ашиглагддаг. Энэ хэсэгт Queue-ийн үр ашигтай ажиллагааг хангахын тулд Lists болон Sets өгөгдлийн бүтцийг хэрхэн ашиглаж болохыг судална. Queue нь Simple Queue (Энгийн дараалал), Circular Queue (Тойрог дараалал), Priority Queue (Эрэмбэтэй дараалал), Double-Ended Queue (Deque) гэсэн үндсэн төрлүүдтэй.

Queue хэрэгжүүлэхэд List vs Set-ийн ялгаа

Шинж чанар	List	Set
FIFO дараалал хадгалах	Тийм	Үгүй
Давтагдсан утга хадгалах	Болно	Болохгүй
Хайх хурд	$O(n)$	$O(1)$
Устгах хурд	$O(n)$	$O(1)$

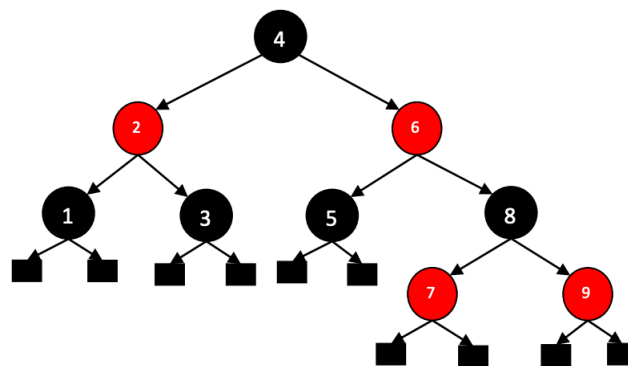
Table 1.12: List болон Sets

### Улаан-хар мод (Red-black tree)

Улаан-хар мод нь хоёртын хайлтын мод (Binary Search tree - BST )-той төстэй бүтэцтэй боловч зангилаа бүрдээ өнгө агуулсан өөрийгөө тэнцвэржүүлдэг, нэг битийн нэмэлт мэдээлэл агуулсан хоёртын хайлтын мод юм. Улаан-хар мод зангилаа нь улаан эсвэл хар өнгөтэй байх ба аль ч зам нь дараалсан хоёр улаан зангилаа агуулахгүй, зам бүр тэнцүү тооны хар зангилаа агуулсан байдаг. Хоёртын хайлтын мод нь дараах шинж чанарыг хангасан бол улаан-хар мод болно:

1. Бүх зангилаа (node) улаан эсвэл хар өнгөтэй байна;

2. Модны үндэс (root) нь үргэлж хар өнгөтэй байна;
3. Улаан зангилааны хүүхдүүд үргэлж хар байна (Red-Red Rule буюу улаан өнгийн хоёр дараалсан зангилаа байж болохгүй);
4. Модны аливаа зам дагуу навч (null node) хүртэлх хар зангилааны тоо ижил байна (Black Height Rule);
5. Шинээр нэмэгдсэн зангилаа улаан өнгөтэй байна.



Зураг 1.12: Улаан-хар мод жишээ

### Давуу тал ба хугацааны комплекс

Улаан-хар мод нь хайлт, оруулах, устгах үйлдлийг  $O(\log n)$  хугацаанд гүйцэтгэх ба энэ нь их хэмжээний өгөгдөл дээр хурдан ажилдаг. AVL модтой харьцуулахад улаан-хар мод нь тэнцвэржүүлэлтийн нөхцөлийг арай зөөлөн баримталдаг тул оруулалт, устгалтын үйлдлүүдийг хурдан гүйцэтгэх боломжтой. AVL мод илүү хатуу тэнцвэрийг баримталдаг бөгөөд энэ нь илүү олон эргүүлэлт хийх шаардлагатай болгодог тул нэмэлт зардал гаргадаг. Улаан-хар мод нь тэнцвэр болон тэнцвэржүүлэх үйлдлийн зардлын хоорондын сайн харьцааг хадгалдаг тул хатуу тэнцвэр шаардлагагүй, үр ашиг чухал тохиолдолд илүү тохиромжтой.

## Хэрэглээ

Улаан-хар модыг олон төрлийн хэрэглээнд өргөнөөр ашигладаг.

### 1. Өгөгдлийн сангийн системүүд

Улаан-хар модыг өгөгдлийн сангийн системүүд индексжүүлэлт болон өгөгдлийг үр ашигтай хайх зориулалтаар ашигладаг. Тэдгээрийн тэнцвэртэй бүтэц нь том хэмжээний өгөгдөл дээр ч хайлтыг хурдан гүйцэтгэх боломжийг олгодог. Жишээлбэл, PostgreSQL болон Oracle зэрэг алдартай өгөгдлийн сангийн удирдлагын системүүд (DBMS) улаан-хар модыг ашиглан өгөгдлийг индексжүүлж, хүсэлтийг оновчтой боловсруулахад хэрэглэдэг.

### 2. Сүлжээний маршрутын алгоритмууд

Улаан-хар мод нь сүлжээний маршрутын алгоритмуудад маш чухал үүрэг гүйцэтгэдэг. Энэ нь хурдан хайлт болон оновчтой маршрутын шийдвэр гаргах боломжийг олгодог бөгөөд ингэснээр өгөгдлийн багцуудыг (packet) үр ашигтай дамжуулж, сүлжээний гүйцэтгэлийг сайжруулдаг. Жишээлбэл, Open Shortest Path First (OSPF) зэрэг интернэт маршрутын протоколууд нь улаан-хар модыг ашиглан маршрутын мэдээллийг хадгалж, оновчтой шийдвэр гаргадаг.

### 3. Програмчлалын хэлний хөрвүүлэгч (Compiler)

Програмчлалын хэлний компиляторууд (compiler) нь симбол хүснэгт (symbol table) хэрэгжүүлэхдээ улаан-хар модыг ашигладаг. Симбол хүснэгт нь хувьсагч, функц, классын мэдээллийг хадгалах үүрэгтэй бөгөөд энэ нь код хөрвүүлэх явцад хаяг оноох, хараат байдлыг шийдвэрлэхэд тусалдаг. LLVM зэрэг компиляторын фреймворкууд улаан-хар модыг ашиглан симбол хүснэгтийг үр ашигтай удирдаж, программын хэлний хөрвүүлэлтийг оновчтой гүйцэтгэдэг.

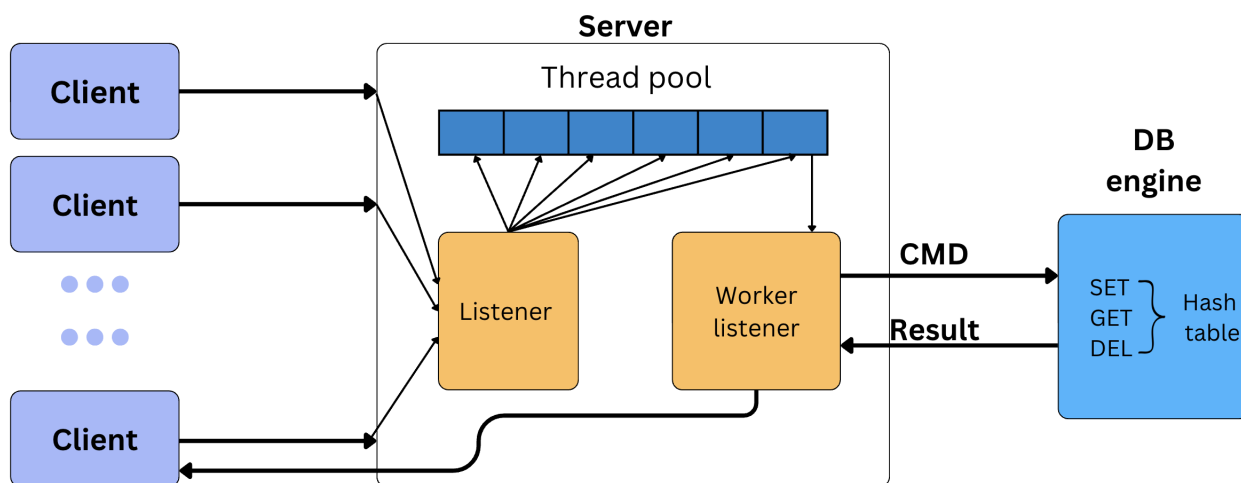
## 1.4 Бүлгийн хураангуй

Санах ойд суурилсан өгөгдлийн сан нь уламжлалт өгөгдлийн сангаас хурдаар илүү ба hash, queue, red-black tree өгөгдлийн бүтцүүд түгээмэл ашиглагддаг. C++ хэл дээр олон урсгалт pool ашиглан санах ойд суурилсан өгөгдлийн санг хэрэгжүүлэх нь олон хэрэглэгчийн хүсэлтийг нэг дор боловсруулах боломжийг олгодог. Мөн түүнчлэн Redis технологи Memcached ба бусад санах ойд суурилсан өгөгдлийн сангуудаас олон төрлийн өгөгдлийн бүтэц ашиглах боломжтой, хурдан, зэрэг олон давуу талтайг судаллаа.

## 2. ШИНЖИЛГЭЭ, ЗОХИОМЖ

### 2.1 Системийн архитектур

Системийн архитектурыг (Зураг 2.1) дүрслэн харуулав. Санах ойд байрших түлхүүр, утга өгөгдлийн сангийн систем нь хэрэглэгч, сервер, өгөгдлийн сан гэсэн үндсэн бүрэлдэхүүн хэсгүүдээс бүрдэнэ. Систем нь түлхүүр-утга өгөгдлийн сангийн серверийг хөгжүүлж, хэрэглэгч найдвартай холбогдож, олон хэрэглэгчийн хүсэлтийг зэрэг боловсруулж, өндөр бүтээмжтэй системийг бүтээхэд суурилсан. Олон хэрэглэгчийн хүсэлтийг TCP сокет, олон хэрэглэгчийн хүсэлтийг Thread pool, өндөр бүтээмж Hash хүснэгт ашиглана тус бүр шийдсэн.

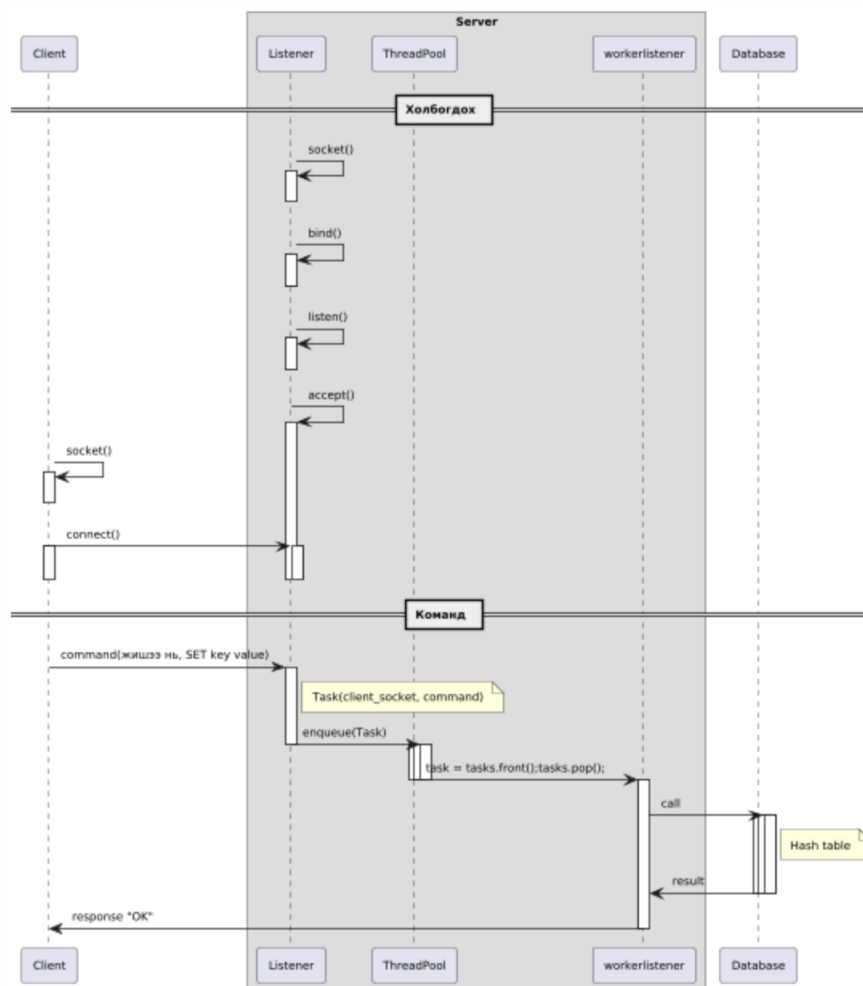


Зураг 2.1: Системийн архитектур

### 2.2 Үйл ажиллагааны дараалал

(Зураг 2.2)-т системийн бүрэлдэхүүн хэсгүүдийн хоорондын харилцан үйлчлэлийн дарааллыг үйл ажиллагааны дарааллын диаграммаар дүрслэн үзүүлэв. Энэхүү диаграмм

нь хэрэглэгч сервертэй холбогдож, команд илгээх, хариу мэдээлэл хүлээн авах үйл явцыг үе шаттайгаар харуулдаг. Диаграммаар дамжуулан хэрэглэгчийн илгээсэн команд хэрхэн серверээр дамжин боловсруулагдаж, шаардлагатай өгөгдөлд хандалт хийсний дараа хариу илгээгддэг үйл ажиллагааны урсгалыг тодорхойлоход чиглэсэн болно.



Зураг 2.2: Үйл ажиллагааны дараалал

## 2.3 Функциональ шаардлага

1. Систем нь хэрэглэгчид түлхүүр утга хосыг оруулах, унших, устгах, шинэчлэх, шалгах, бүх түлхүүрийг жагсаах үйлдлүүдийг SET, GET, DEL, UPD, HAS, KEYS команд

ашиглан гүйцэтгэдэг байх;

2. Систем нь нэгэн зэрэг олон хэрэглэгчийн хүсэлтийг боловсруулдаг байх;
3. Систем нь командын форматыг шалгах, баталгаажуулах үндсэн шалгууртай байх;
4. Систем нь бүх хэрэглэгчдэд USE, DBNAME, DBLIST, EXIT команд ашиглан өгөгдлийн санг солих, харах, жагсаах холболтоо таслах боломжтой байх;
5. Систем нь хэрэглэгчид алдаатай өгөгдөл оруулбал харгалзах алдааны зурвасыг буцаадаг байх;
6. Систем нь тодорхой нэг дугаартай (порттой) холбогдож ажиллах боломжтой байх;
7. Систем нь хэрэглэгчид алдаатай өгөгдөл оруулахаас сэргийлж командыг бичих интерфейс харуулдаг байх.

## **2.4 Функциональ бус шаардлага**

1. Хурдан байх;
2. Цэгцтэй, ойлгомжтой байх;
3. Хоцролт багатай байх;
4. Ямар ч үед хэвийн ажиллагаатай байх;
5. Гацалт үүсдэггүй байх.

## 3. ХЭРЭГЖҮҮЛЭЛТ

### 3.1 Сервер

Серверийн хэрэгжүүлэлтийг дээр дурдсан архитектур болон үйл ажиллагааны дарааллын дагуу боловсруулсан бөгөөд хоорондын уялдаа, хамаарал, үүрэг болон гол үйлдлүүдийг нарийвчлан тодорхойлсон.

#### **class ConnectionListener**

**Үндсэн үүрэг:** Серверийн socket үүсгэж, IP хаяг болон портыг холбон, хэрэглэгч холбогдох үед шинэ холболт үүсгэн, тухайн холболтыг даалгавар болгон ажлын дараалал руу нэмэх.

**Гол үйлдлүүд:**

- `server_fd = socket(AF_INET, SOCK_STREAM, 0);`
- `bind(server_fd, (struct sockaddr*)&address, sizeof(address));`
- `listen(server_fd, 10);`
- `client_socket = accept(server_fd, (struct sockaddr*)&address, (socklen_t*)&addrlen);`
- `pool.enqueue(Task(client_socket, command));`

#### **struct Task**

**Үндсэн үүрэг:** Хэрэглэгчийн илгээсэн команд болон тухайн хэрэглэгчийн сокетын мэдээллийг багцлан ажлын дараалалд хүлээлгэн өгөх.

**Гол үйлдэл:**

- `Task(client_socket, command);`



## **class ThreadPool**

**Үндсэн үүрэг:** Ажлын дарааллаас даалгавруудыг авч, тэдгээрийг processCommand() функцээр боловсруулан, гарсан үр дүнг холбогдох хэрэглэгчид сокет ашиглан илгээх.

**Гол үйлдлүүд:**

- tasks.pop();
- std::string result = processCommand(task.client\_socket, task.command);
- send(task.client\_socket, result.c\_str(), result.length(), 0);

## **processCommand()**

**Үндсэн үүрэг:** Хэрэглэгчийн илгээсэн командыг боловсруулж, өгөгдлийн сангийн холбогдох үйлдлийг гүйцэтгэнэ. Энэ функц нь KVDatabase болон DatabaseManager зэрэг бүрэлдэхүүнтэй хамтран ажиллана.

## **class KVDatabase**

**Үндсэн үүрэг:** Нэг өгөгдлийн сангийн хүрээнд түлхүүр-утга хосуудыг хадгалах болон боловсруулах.

**Гол үйлдлүүд:**

- SET() – Тодорхой түлхүүрт утга хадгалах;
- GET() – Түлхүүрээр утга хайж авах;
- HAS() – Түлхүүр байгаа эсэхийг шалгах;
- UPD() – Түлхүүрийн утгыг шинэчлэх;
- DEL() – Түлхүүр устгах;
- KEYS() – Бүх түлхүүрүүдийн жагсаалтыг буцаах.

## **class DatabaseManager**

**Үндсэн үүрэг:** Олон өгөгдлийн санг зэрэг удирдаж, хэрэглэгч тус бүрийн хувийн өгөгдлийн сантай ажиллах боломжийг хангах.

**Гол үйлдлүүд:**

- `getDatabase(name)` – Нэрээр өгөгдлийн санг олж авах;
- `listDatabases()` – Бүх өгөгдлийн сангуудын нэрийг жагсаах.

```
1 #include <iostream>
2 #include <unordered_map>
3 #include <string>
4 #include <sstream>
5 #include <vector>
6 #include <queue>
7 #include <thread>
8 #include <mutex>
9 #include <condition_variable>
10 #include <atomic>
11 #include <functional>
12 #include <cstring>
13 #include <sys/socket.h>
14 #include <netinet/in.h>
15 #include <arpa/inet.h>
16 #include <unistd.h>
17
18 class KVDatabase {
19 private:
20     std::unordered_map<std::string, std::string> store;
```

```

21     std::mutex mutex;
22
23 public:
24     bool set(const std::string& key, const std::string& value) {
25         std::lock_guard<std::mutex> lock(mutex);
26         store[key] = value;
27         return true;
28     }
29
30     bool get(const std::string& key, std::string& value) {
31         std::lock_guard<std::mutex> lock(mutex);
32         if (store.find(key) != store.end()) {
33             value = store[key];
34             return true;
35         }
36         return false;
37     }
38     bool has(const std::string& key) {
39         std::lock_guard<std::mutex> lock(mutex);
40         return store.find(key) != store.end();
41     }
42
43     bool upd(const std::string& key, const std::string& new_value) {
44         std::lock_guard<std::mutex> lock(mutex);
45         auto it = store.find(key);
46         if (it != store.end()) {
47             it->second = new_value;
48             return true;

```

```

49     }
50     return false;
51 }
52
53 bool del(const std::string& key) {
54     std::lock_guard<std::mutex> lock(mutex);
55     if (store.find(key) != store.end()) {
56         store.erase(key);
57         return true;
58     }
59     return false;
60 }
61
62 std::vector<std::string> keys() {
63     std::lock_guard<std::mutex> lock(mutex);
64     std::vector<std::string> result;
65     for (const auto& pair : store) {
66         result.push_back(pair.first);
67     }
68     return result;
69 }
70 };
71
72 class DatabaseManager {
73 private:
74     std::unordered_map<std::string, KVDatabase> databases;
75     std::mutex mutex;
76

```

```

77 public:
78     KVDatabase& getDatabase(const std::string& name) {
79         std::lock_guard<std::mutex> lock(mutex);
80         return databases[name];
81     }
82     std::vector<std::string> listDatabases() {
83         std::lock_guard<std::mutex> lock(mutex);
84         std::vector<std::string> names;
85         for (const auto& pair : databases) {
86             names.push_back(pair.first);
87         }
88         return names;
89     }
90
91 };
92
93 struct Task {
94     int client_socket;
95     std::string command;
96     Task(int socket, const std::string& cmd) : client_socket(socket),
97         command(cmd) {}
98
99 };
100
101 class ThreadPool {
102 private:
103     std::vector<std::thread> workers;
104     std::queue<Task> tasks;
105     std::mutex queue_mutex;

```

```

104     std::condition_variable condition;
105     std::atomic<bool> stop;
106     DatabaseManager& dbManager;
107     std::unordered_map<int, std::string> clientDBs;
108     std::mutex dbMapMutex;
109
110     std::vector<std::string> split(const std::string& s, char delimiter
111         ) {
112         std::vector<std::string> tokens;
113         std::string token;
114         std::istringstream tokenStream(s);
115         while (std::getline(tokenStream, token, delimiter)) {
116             tokens.push_back(token);
117         }
118         return tokens;
119     }
120
121     std::string processCommand(int client_socket, const std::string&
122         command) {
123         std::vector<std::string> tokens = split(command, ' ');
124         if (tokens.empty()) {
125             return "ERROR: Empty command\n";
126         }
127
128         if (tokens[0] == "USE" && tokens.size() == 2) {
129             std::lock_guard<std::mutex> lock(dbMapMutex);
130             clientDBs[client_socket] = tokens[1];
131             return "OK: Switched to database " + tokens[1] + "\n";

```

```

130     }
131
132     std::string dbname = "db";
133     {
134         std::lock_guard<std::mutex> lock(dbMapMutex);
135         if (clientDBs.count(client_socket)) {
136             dbname = clientDBs[client_socket];
137         }
138     }
139
140     KVDatabase& db = dbManager.getDatabase(dbname);
141
142     if (tokens[0] == "SET" && tokens.size() >= 3) {
143         std::string value = tokens[2];
144         for (size_t i = 3; i < tokens.size(); i++) {
145             value += " " + tokens[i];
146         }
147
148         db.set(tokens[1], value);
149         return "OK\n";
150     } else if (tokens[0] == "GET" && tokens.size() == 2) {
151         std::string value;
152         if (db.get(tokens[1], value)) {
153             return "VALUE: " + value + "\n";
154         } else {
155             return "NULL: Key not found\n";
156         }
157     } else if (tokens[0] == "DEL" && tokens.size() == 2) {

```

```

158         if (db.del(tokens[1])) {
159             return "OK: Key deleted\n";
160         } else {
161             return "ERROR: Key not found\n";
162         }
163     } else if (tokens[0] == "HAS" && tokens.size() == 2) {
164         if (db.has(tokens[1])) {
165             return "YES: Key has\n";
166         } else {
167             return "NO: Key not found\n";
168         }
169     } else if (tokens[0] == "UPD" && tokens.size() >= 3) {
170         std::string new_value = tokens[2];
171         for (size_t i = 3; i < tokens.size(); i++) {
172             new_value += " " + tokens[i];
173         }
174         if (db.upd(tokens[1], new_value)) {
175             return "OK: Value update\n";
176         } else {
177             return "ERROR: Key not found\n";
178         }
179     } else if (tokens[0] == "DBNAME" && tokens.size() == 1) {
180         std::string dbname = "db";
181         {
182             std::lock_guard<std::mutex> lock(dbMapMutex);
183             if (clientDBs.count(client_socket)) {
184                 dbname = clientDBs[client_socket];
185             }

```



```

186         }
187         return "CURRENT DB: " + dbname + "\n";
188     }
189     else if (tokens[0] == "DBLIST" && tokens.size() == 1) {
190         std::vector<std::string> dbs = dbManager.listDatabases();
191         std::string result = "DATABASES:\n";
192         for (const auto& name : dbs) {
193             result += "- " + name + "\n";
194         }
195         return result;
196     }
197
198
199     else if (tokens[0] == "KEYS" && tokens.size() == 1) {
200         std::vector<std::string> keys = db.keys();
201         std::string result = "KEYS:\n";
202         for (const auto& key : keys) {
203             result += "- " + key + "\n";
204         }
205         return result;
206     } else {
207         return "ERROR: Unknown command or wrong format\n";
208     }
209 }
210
211 public:
212     ThreadPool(size_t threads, DatabaseManager& manager) : stop(false),
        dbManager(manager) {

```

```

213     for (size_t i = 0; i < threads; ++i) {
214         workers.emplace_back([this] {
215             while (true) {
216                 Task task(0, "");
217                 {
218                     std::unique_lock<std::mutex> lock(queue_mutex);
219                     condition.wait(lock, [this] { return stop || !
220                                     tasks.empty(); });
221                     if (stop && tasks.empty()){
222                         return;
223                     }
224                     task = tasks.front();
225                     tasks.pop();
226                     std::string result = processCommand(task.
227                                                         client_socket, task.command);
228                     send(task.client_socket, result.c_str(), result.
229                         length(), 0);
230                 }
231             });
232         }
233     }
234
235     void enqueue(Task task) {
236         {
237             std::unique_lock<std::mutex> lock(queue_mutex);
238             tasks.push(task);
239         }
240     }

```

```

238         condition.notify_one();
239     }
240
241     ~ThreadPool() {
242     {
243         std::unique_lock<std::mutex> lock(queue_mutex);
244         stop = true;
245     }
246     condition.notify_all();
247     for (std::thread& worker : workers) {
248         worker.join();
249     }
250 }
251 };
252
253 class ConnectionListener {
254 private:
255     int server_fd;
256     int port;
257     ThreadPool& pool;
258     std::atomic<bool> running;
259     std::thread listener_thread;
260
261 public:
262     ConnectionListener(int port_num, ThreadPool& thread_pool)
263         : port(port_num), pool(thread_pool), running(false) {}
264
265     bool start() {

```

```

266     server_fd = socket(AF_INET, SOCK_STREAM, 0);
267     if (server_fd < 0) return false;
268
269     int opt = 1;
270     setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(
        opt));
271
272     struct sockaddr_in address;
273     address.sin_family = AF_INET;
274     address.sin_addr.s_addr = INADDR_ANY;
275     address.sin_port = htons(port);
276
277     if (bind(server_fd, (struct sockaddr*)&address, sizeof(address)
        ) < 0) return false;
278     if (listen(server_fd, 10) < 0) return false;
279
280     running = true;
281     listener_thread = std::thread([this, address]() {
282         while (running) {
283             int addrlen = sizeof(address);
284             int client_socket = accept(server_fd, (struct sockaddr
                *)&address, (socklen_t*)&addrlen);
285             if (!running || client_socket < 0) continue;
286             std::cout << "Client connected" << std::endl;
287             std::thread([this, client_socket]() {
288                 char buffer[1024] = {0};
289                 while (true) {
290                     memset(buffer, 0, sizeof(buffer));

```

```

291         int bytes_read = read(client_socket, buffer,
292                                sizeof(buffer));
293         if (bytes_read <= 0) {
294             close(client_socket);
295             return;
296         }
297         std::string command(buffer);
298         if (!command.empty() && command.back() == '\\n')
299             command.pop_back();
300         if (command == "EXIT") {
301             send(client_socket, "Goodbye!\\n", 9, 0);
302             close(client_socket);
303             return;
304         }
305         pool.enqueue(Task(client_socket, command));
306     }
307     }).detach();
308 }
309
310
311 void stop() {
312     running = false;
313     int sock = socket(AF_INET, SOCK_STREAM, 0);
314     struct sockaddr_in addr;
315     addr.sin_family = AF_INET;
316     addr.sin_port = htons(port);

```

```

317     addr.sin_addr.s_addr = inet_addr("127.0.0.1");
318     connect(sock, (struct sockaddr*)&addr, sizeof(addr));
319     close(sock);
320
321     if (listener_thread.joinable()) {
322         listener_thread.join();
323     }
324     close(server_fd);
325 }
326
327 ~ConnectionListener() {
328     stop();
329 }
330 };
331
332 int main(int argc, char* argv[]) {
333     int port = 8080;
334     if (argc > 1) port = std::stoi(argv[1]);
335
336     const int NUM_THREADS = 4;
337     DatabaseManager dbManager;
338     ThreadPool pool(NUM_THREADS, dbManager);
339     ConnectionListener listener(port, pool);
340
341     if (!listener.start()) {
342         std::cerr << "Failed to start listener" << std::endl;
343         return -1;
344     }

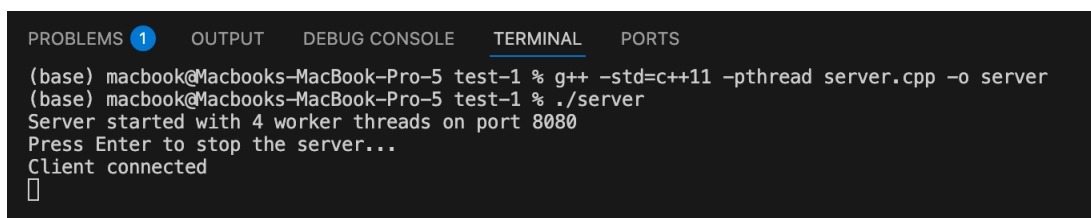
```

```

345
346     std::cout << "Server started with " << NUM_THREADS << " worker
        threads on port " << port << std::endl;
347     std::cout << "Press Enter to stop the server..." << std::endl;
348     std::cin.get();
349
350     listener.stop();
351     std::cout << "Server stopped" << std::endl;
352     return 0;
353 }

```

Код 3.1: Серверийн хэрэгжүүлэлт



```

PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL PORTS
(base) macbook@Macbooks-MacBook-Pro-5 test-1 % g++ -std=c++11 -pthread server.cpp -o server
(base) macbook@Macbooks-MacBook-Pro-5 test-1 % ./server
Server started with 4 worker threads on port 8080
Press Enter to stop the server...
Client connected

```

Зураг 3.1: Сервер

## 3.2 Клиент

Клиент хэрэгжүүлэлт нь TCP сокет ашиглан сервертэй холбогдож, хэрэглэгчийн оруулсан командыг сервер рүү илгээж, серверээс ирсэн хариуг уншиж харуулдаг.

## Гол үйлдлүүд:

- sock = socket(AF\_INET, SOCK\_STREAM, 0);
- connect(sock, (struct sockaddr\*)&serv\_addr, sizeof(serv\_addr)) < 0;
- send(sock, input.c\_str(), input.length(), 0);
- valread = read(sock, buffer, sizeof(buffer));

```
1 #include <iostream>
2 #include <string>
3 #include <cstring>
4 #include <sys/socket.h>
5 #include <netinet/in.h>
6 #include <arpa/inet.h>
7 #include <unistd.h>
8
9
10 void printHelp() {
11     std::cout << "\t\t n:" << std::endl;
12     std::cout << "  SET <key> <value> -          " << std::endl;
13     std::cout << "  GET <key> -          " << std::endl;
14     std::cout << "  DEL <key> -          " << std::endl;
15     std::cout << "  UPD <key><value> -          " << std::endl;
16     std::cout << "  HAS <key> -          " << std::
17         endl;
18     std::cout << "  KEYS -          " << std
19         ::endl;
20     std::cout << "  USE -          database - .
21         " << std::endl;
```



```

19     std::cout << "    DBNAME                -                database -        .
        " << std::endl;
20     std::cout << "    DBLIST                -                database -        " << std
        ::endl;
21     std::cout << "    EXIT                -                " << std::endl;
22 }
23
24 int main() {
25     const char* server_ip = "127.0.0.1";
26     const int port = 8080;
27
28     int sock = socket(AF_INET, SOCK_STREAM, 0);
29     if (sock < 0) {
30         std::cerr << "Socket creation error\n";
31         return 1;
32     }
33
34     struct sockaddr_in serv_addr;
35     serv_addr.sin_family = AF_INET;
36     serv_addr.sin_port = htons(port);
37
38     if (inet_pton(AF_INET, server_ip, &serv_addr.sin_addr) <= 0) {
39         std::cerr << "Invalid or unsupported IP address\n";
40         return 1;
41     }
42
43     if (connect(sock, (struct sockaddr*)&serv_addr, sizeof(serv_addr))
        < 0) {

```

```

44     std::cerr << "Failed to server\n";
45     return 1;
46 }
47
48 std::cout << "Connected to the server. Please enter a command:\n";
49 printHelp();
50
51 while (true) {
52     std::string input;
53     std::cout << "db> ";
54     std::getline(std::cin, input);
55
56     if (input.empty()) continue;
57
58     input += "\n";
59     send(sock, input.c_str(), input.length(), 0);
60
61     if (input.find("EXIT") != std::string::npos) {
62         break;
63     }
64
65     char buffer[1024] = {0};
66     int valread = read(sock, buffer, sizeof(buffer));
67     if (valread > 0) {
68         std::cout << buffer;
69     } else {
70         std::cerr << "Failed to server\n";
71         break;

```

```

72     }
73 }
74
75 close(sock);
76 std::cout << "Connection closed.\n";
77 return 0;
78 }

```

Код 3.2: Клиентийн хэрэгжүүлэлт

```

• (base) macbook@Macbooks-MacBook-Pro-5 test-2 % g++ -std=c++11 -pthread client.cpp -o client
○ (base) macbook@Macbooks-MacBook-Pro-5 test-2 % ./client
Connected to the server. Please enter a command:

Командууд:
SET <key> <value> - Түлхүүр утга хосыг хадгалах
GET <key>         - Түлхүүрээр утгыг олж авах
DEL <key>         - Түлхүүр утгын хосыг устгах
UPD <key><value>  - Түлхүүрээр утгыг өөрчлөх
HAS <key>        - Түлхүүр өгөгдлийн санд байгааг харах
KEYS              - Өгөгдлийн сангийн бүх түлхүүрүүдийг жагсаах
USE              - Хэрэглэгч өөрийн ашиглаж буй database-аа солих.
DBNAME           - Тухайн хэрэглэгчийн идэвхтэй database-г харуулах.
DBLIST           - Бүх database-ийн нэрсийг жагсаах
EXIT             - Гаргах
db> 

```

Зураг 3.2: Клиент

### 3.3 Үр дүн

#### 3.3.1 Хугацааны харьцуулалт

	1'000'000 өгөгдөл "SET"	1'000'000 хэрэглэгч параллель
Миний систем	77.1359	0.1476
Redis	71.6279	11.5743

Table 3.1: Хугацааны харьцуулалт

# Дүгнэлт

Бакалаврын судалгааны ажлын хүрээнд санах ойд суурилсан түлхүүр-утга өгөгдлийн сангийн сервер болон клиент программыг C++ программчлалын хэл ашиглан амжилттайгаар хөгжүүлж, онолын мэдлэгээ практикт хэрэгжүүлэн бататгасан. Судалгааны явцад зөвхөн программчлалын техник арга зүйг эзэмшихээс гадна асуудлыг шат дараатайгаар задлан шинжлэх, оновчтой шийдэл боловсруулах, хөгжүүлэлтийн үйл явцыг зохион байгуулах, мөн албан ёсны баримт бичиг болон эх сурвалжуудыг үр өгөөжтэйгөөр ашиглах зэрэг мэргэжлийн чухал ур чадваруудыг эзэмшсэн болно.

Судалгааны ажлын явцад тулгарсан техникийн болон зохион байгуулалтын шинжтэй хүндрэлүүдийг оновчтой төлөвлөлт, үе шаттай гүйцэтгэлээр шийдвэрлэх явцад мэргэжлийн чадамжийг улам сайжруулах нөхцөл бүрдсэн.

Цаашид уг судалгааны ажил болон түүний хүрээнд олж авсан мэдлэг, туршлагадаа үндэслэн мэргэжлийн ур чадвараа улам гүнзгийрүүлэн хөгжүүлж, программ хангамжийн салбарт олон улсын түвшинд ажиллах чадвартай инженер болохыг зорьж байна. Энэхүү судалгааны ажил нь миний мэргэжлийн замналын эхлэл бөгөөд цаашид илүү ихийг суралцах, тасралтгүй хөгжих шаардлагатайг ухамсарлан ойлгосон болно.

# Bibliography

- [1] Younes Khourdifi Alae El Alami, Mohamed Bahaj. Supply of a key value database redis in-memory by data from a relational database. *2018 19th IEEE Mediterranean Electrotechnical Conference (MELECON)*, pages 46–51, May 2018.
- [2] Michael Cornwell. Anatomy of a solid-state drive. *Queue*, 10(10):30–36, Oct 2012.
- [3] Qian Liu. A high performance memory key-value database based on redis. *Journal of Computers*, 14(3):170–183, 2019.
- [4] Ramya S. On demand cache management and cache migration to balance the cache load. *International Journal for Research in Applied Science and Engineering Technology*, V(IX):831–833, Sep 2017.
- [5] Barak Shoshany. A c++17 thread pool for high-performance scientific computing. *SoftwareX*, 26:101687, May 2024.
- [6] D. Puida undefined undefined. Modification of a thread pool algorithm with multiple task queues. *Computer systems and network*, 5(1):96–102, Dec 2023.
- [7] Dai ZJ. disk.frame: Larger-than-ram disk-based data manipulation framework. *CRAN: Contributed Packages*, Aug 2019.