

Genomorientierte Bioinformatik

-

BamFeatures

Malte Weyrich

DECEMBER 2024

Das Sequenzieren in der Bioinformatik generiert Milliarden von *Reads* pro *Sample*, welche mit einem *Mapper* an das *Referenzgenom* aligniert werden. Diese Daten werden in einer *Sequence Alignment Map (SAM)* Datei gespeichert und können zusätzlich in ein komprimiertes Format, einer *Binary Alignment Map (BAM)* Datei, umgewandelt werden. Anschließend können verschiedene Analysen auf den *BAM* Dateien durchgeführt werden. Die in diesem Report diskutierte *JAR* liest eine gegebene *paired-end RNA-seq BAM* Datei ein und errechnet verschiedene Features, die in einer *<tsv>* Datei gespeichert werden. Die JAR wurde auf drei verschiedenen *BAM* Dateien ausgeführt, um damit anschließend die *RPKM* Werte zu berechnen. Zudem wird die *JAR* anhand ihrer Laufzeit und Korrektheit analysiert.

Inhalt

1 RNA-seq	3
2 Java Programm	3
2.1 Argumente	4
2.2 Logik	4
2.2.1 Erstellen der ReadPair Objekte	4
2.2.2 Berechnung der ReadPair Attribute	5
2.2.3 Reduktion der Heap Benutzung	8
2.3 Korrektheit	9
3 Ergebnisse	9
3.1 Laufzeit	10
3.2 Benchmarking	10

1 – RNA-seq

Die Bezeichnung *RNA-seq* bezieht sich auf ein bestimmtes Sequenzierprotokoll der Bioinformatik bei dem möglichst alle exprimierten Transkripte mehrerer Samples mittels Hochdurchsatzsequenziergeräten wie *Illumina* verarbeitet und die resultierenden *Reads* in Ausgabe Dateien abgespeichert werden. Bei *Illumina* werden die extrahierten Transkripte mittels Ultraschall oder enzymatischer Fermentation in Fragmente mit ähnlicher Länge zerstückelt und mit Adaptersequenzen und Barcodes versehen. Darauf folgt eine *PCR* Amplifikation der Fragmente, um das Signal zu verstärken. Die amplifizierten Fragmente können nun im Sequenzzyklus von *Illumina*, Base für Base, gelesen werden. Dabei wird jedoch nicht das ganze Fragment gelesen, sondern jeweils nur ca. 100BP (abhängig von Voreinstellung und Protokoll) beider Enden des Fragments (*paired-end sequencing*). Somit entstehen pro Fragment zwei *Reads*, ein forward *Read* und ein reverse *Read*, welche zu einem *ReadPair* zusammengefasst werden können. In dem Protokoll von *Illumina* werden zuerst alle *Reads* eines Endes aller Fragmente gemacht, dann wird das Fragment, **vereinfacht gesagt**, auf der *Flow Zelle* umgedreht (durch Replikation), wodurch die umgedrehten Fragmente jeweils das Komplement ihres ursprünglichen Fragments sind. Somit sollten die *Reads* eines *ReadPairs* immer auf entgegengesetzte Strände ("+/-") *mappen*. Ist bei einem Sequenzierexperiment die Ausgangskonfiguration der Fragmente bekannt, so handelt es sich um ein *strangspezifisches* Experiment und man kann allen *Reads* einem festen Strang zuordnen, je nach dem, ob das Fragment in der Anfangskonfiguration vom "-" oder vom "+" Strang kam. Ein *Mapper* würde nun solche *ReadPairs* an einem *Referenzgenom mappen* und eine (oder mehrere) *SAM/BAM* Datei(en) erstellen, welche unter anderem die Koordinaten der alignierten *Reads* basierend auf dem *Referenzgenom* beinhalten. Die Alignment Daten dieser Datei können nun von der *BamFeatures JAR* weiter annotiert werden.

2 – Java Programm

Usage:

```
java -jar bam.jar -bam <bamPath> -o <outputPath> -gtf <gtfPath> \\
      [-frstrand <true/false>] [-lengths]
```

2.1. Argumente

Zusätzlich zu den vorausgesetzten Argumenten (*-bam*, *-o*, *-gtf*) können noch *-frstrand* und *-lengths* angegeben werden. Bei einem Strangpositiven Experiment (*-frstrand true*) ist der forward *Read* auf dem "+" und der reverse *Read* auf dem "-". So kann die *JAR* die *Reads* korrekt zuordnen. Falls die Strangrichtung nicht angegeben ist, werden für beide *Reads* jeweils beide Stränge betrachtet. Die *-lengths* Option wird für die Berechnung der *RPKM* Werte benötigt. Ist diese Option gesetzt, so werden die für die Längennormalisierung benötigte Genlängen der kombinierten Exons jedes Gens berechnet und in einer *<tsv>* Datei gespeichert.

Tabelle 1 – Übersicht der verwendeten *BAM* Dateien

Spezies	BAM	GTF	frstrand
Homo Sapiens	ebna_hisat	GRCh37.75	"/" / "true"
Homo Sapiens	hes_star	GRCh37.75	"/" / "false"
Hefe	nookaew_cm	R64-1-1.75	×

2.2. Logik

2.2.1 Erstellen der *ReadPair* Objekte

Die Einträge der *BAM* Datei werden der Reihe nach von einem *SAMFilereader* eingelesen. Da es sich um *paired-end* Daten handelt, muss für jeden *Read* sein zugehöriger *Mate* gefunden werden. *Reads* die nicht gepaart sind, oder nicht den Qualitätsanforderungen der Aufgabenstellung entsprechen, werden ignoriert. *Read Objekte* die zum ersten Mal vorkommen, werden in einer *HashMap<Id, Read>* *seenEntries* gespeichert und für jede Iteration wird überprüft, ob wir die *Read Id* bereits gesehen haben. Sobald wir zwei zusammengehörende *Reads* identifiziert haben, wird ein neues *ReadPair Objekt* erstellt. Dabei wird die Strangrichtung beim erstellen des Objekts berücksichtigt und die *Reads* jeweils nach *forward* und *reverse* kategorisiert. Zusätzlich werden die *AlignmentBlocks* beider *Reads* zu einem gemeinsamen *Regionvector meltedBlocks* und zwei einzelnen *Regionvectors* (*regionVecFw*, *regionVecRw*) verschmolzen. Dies ist notwendig für die anschließenden Berechnungen und fängt einige *Edge Cases* ab. Hat ein *ReadPair R* z.B. $b_1 \in regionVecFw, b_2 \in regionVecRw$ und $b_1.end == b_2.start - X, X > 0$, so werden diese Blöcke verschmolzen zu: $b_1, b_2 \rightarrow_{melt()} b_{neu}$, mit $b_{neu}.start == b_1.start \wedge b_{neu}.end == b_2.end$.

2.2.2 Berechnung der ReadPair Attribute

Als erstes werden die *igenes* und *cgenes* berechnet:

1. $cgenes := \{g \in cgenes \mid g_{start} < fwRead_{start} \wedge g_{end} > rwRead_{end}\}$
2. $igenes := \{g \in cgenes \mid g_{start} \geq fwRead_{start} \wedge g_{end} \leq rwRead_{end}\}$

Für die Berechnung dieser Attribute verwenden wir ein verschachteltes Objekt *intervalTreeMap* *HashMap<String, HashMap<Boolean, IntervalTree<Gene>>>* verwendet, welches für jedes Chromosom die Gene nach ihrem Strang in Intervalbäumen abgespeichert hat (Strang ist entweder: `[true|false|null]`). Falls $|cgenes| == 0$ aber $|igenes| > 0$ wird das *ReadPair* verworfen und mit dem nächsten weiter gemacht, sind beide Mengen leer, so wird die kürzeste Distanz zu benachbarten Genen ausgerechnet und in *gdist* abgespeichert. Danach wird das *ReadPair* auf *split-inconsistency* überprüft (Algorithmus 1), d.h. falls es eine überlappende Region beider *Reads* gibt, müssen die potentiell implizierten Introns beider *Reads* übereinstimmen. Nach dem Aufruf von *getNsplit()* wird

Algorithm 1 getNsplit()

```

1: Input: fw, rw                                ▷ forward and reverse reads
2: if |fw.Blocks| = 1 and |rw.Blocks| = 1 then      ▷ Checks if the reads imply introns
3:   return 0
4: end if
5: overlap = determineOverlap(fw, rw)            ▷ Determine overlap region
6: iFwRegions ← {}                               ▷ Set for containing fw Introns
7: iRwRegions ← {}                               ▷ Set for containing rw Introns
8: iRegions ← {}                                ▷ Set for containing all Introns
9: extractIntronsInOverlap(overlap.x1, overlap.x2, iFwRegions, iRegions, fw)
10: extractIntronsInOverlap(overlap.x1, overlap.x2, iRwRegions, iRegions, rw)
11: if |iRwRegions| ≠ |iFwRegions| then           ▷ split-inconsistent
12:   return -1
13: end if
14: if iRwRegions = iFwRegions then             ▷ Return unique Introns in overlap
15:   return |iRegions|
16: end if
17: return -1                                    ▷ split-inconsistent (default)

```

bei einem *return value* von -1 das *ReadPair* als "split-inconsistent" vermerkt und in die Ausgabedatei übernommen, ansonsten wird die Größe der Menge *iRegions* in der Variable *nsplit* gespeichert und das *ReadPair* weiter prozessiert.

Als nächstes wird das *ReadPair* anhand drei Kategorien annotiert (Abbildung 1):

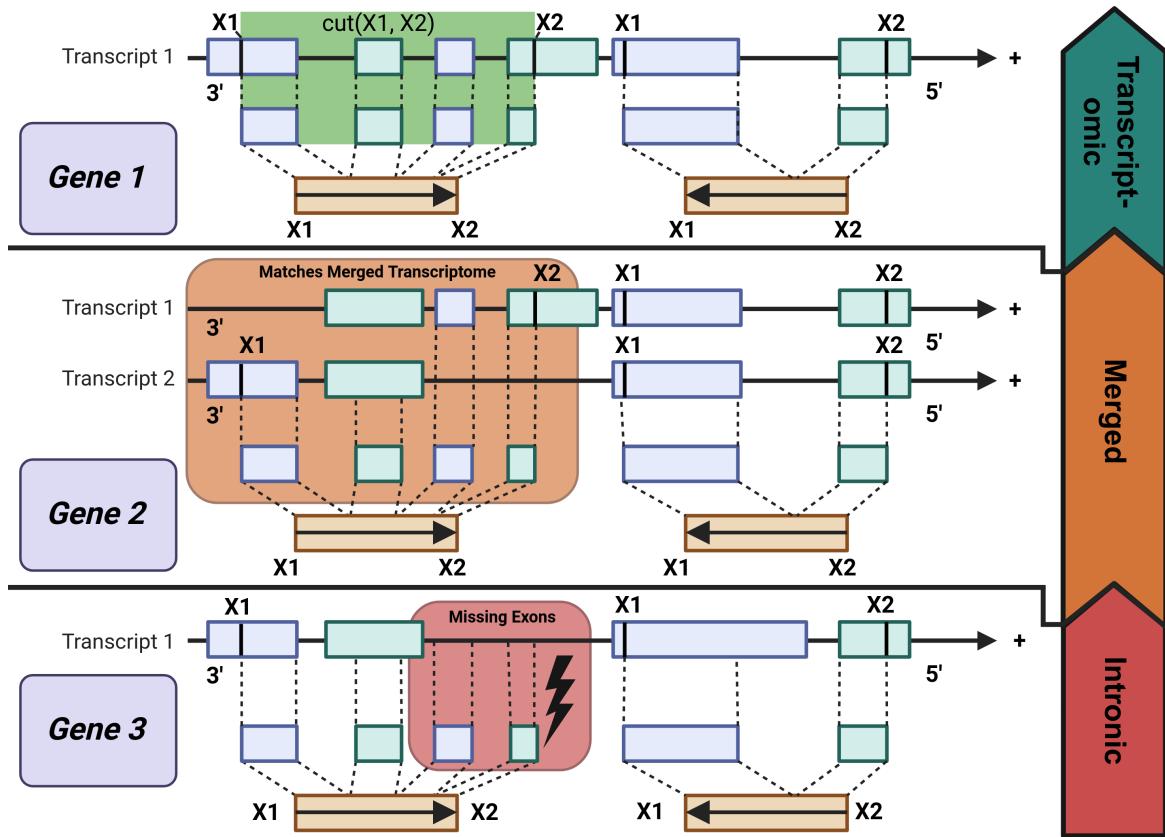


Abbildung 1 – Kategorisierung der *ReadPair*-Regionen in die Klassen „*Transcriptomic*“, „*Merged*“ und „*Intronic*“, wobei eine Priorisierung nach der Reihenfolge „*Transcriptomic*“ > „*Merged*“ > „*Intronic*“ erfolgt. Die *AlignmentBlocks* der *Reads* werden mit den Exons der Transkripte des inkludierenden Gens abgeglichen. Die Abbildung wurde mit [BioRender 2024](#) erstellt.

Bei der Regions-Annotation wird als erstes die ***Transcriptomic*** Kategorie überprüft. Hierbei wird über alle Gene die das *ReadPair* inkludieren iteriert und für jeden *Read* des *ReadPairs* mit der Methode $\text{cut}(X1, X2)$ zwei neue *Regionvectors* aus jedem Transkript ausgeschnitten (Abbildung 1), wobei *X1* der *Alignmentstart* und *X2* das *Alignmentende* des momentanen *Reads* ist. Falls beide ausgeschnittenen *Regionvectors* aus dem Transkript gleich der geschmolzenen *Regionvectors* der *Reads* entsprechen, ist das *ReadPair* ***Transcriptomic*** und das entsprechende Transkript wird zusammen mit dem Gen in die Lösungsmenge genommen.

Für die ***Merged*** Kategorie reicht es, wenn die *Regionvectors* der *Reads* in dem geschmolzenen Transkriptom eines Gens enthalten sind (siehe Abbildung 1). Das geschmolzene Transkriptom wird in Algorithmus 2 berechnet und besteht aus allen annotierten Exons eines Gens. Wenn

ein *ReadPair* dieser Kategorie zugeteilt wird, wird das Gen in einer separaten Lösungsmenge gespeichert. Falls die oberen zwei Ansätze beide nicht zutreffen, so handelt es sich um ein **Intrinsic ReadPair**.

Algorithm 2 Melt Exons into Regions

```

1: Input: transcriptList (list of transcripts containing exons)
2: allExons ← ∅
3: for each transcript in transcriptList do
4:     allExons.addAll(transcript.getExonList())      ▷ Add all exons from transcript to the list
5: end for
6: Sort allExons by start position                      ▷ Sort exons by their start positions
7: meltedRegions ← new TreeSet()                      ▷ Create a new set for melted regions
8: if allExons is not empty then
9:     first ← allExons.get(0)                          ▷ Get the first exon
10:    current ← new Region(first.getStart(), first.getStop()) ▷ Create a region for the first exon
11:    for i ← 1 to allExons.size() − 1 do
12:        exon ← allExons.get(i)
13:        if exon.getStart() ≤ current.getStop() + 1 then
14:            current.setStop(max(current.getStop(), exon.getStop())) ▷ Extend the region if
           exons overlap or are adjacent
15:        else
16:            meltedRegions.add(current)                  ▷ Add the completed region to the set
17:            current ← new Region(exon.getStart(), exon.getStop()) ▷ Start a new region
18:        end if
19:    end for
20:    meltedRegions.add(current)                      ▷ Add the last region
21: end if
22: return meltedRegions                                ▷ Return the melted regions

```

Nach der Regions-Annotation wird der *gcount* mit der Anzahl an annotierten Genen der plausibelsten Kategorie überschrieben. Als nächstes überprüft die *JAR* die Anzahl an *mismatches*

der alignierten *Reads* und die Anzahl an *clipped bases*. *Clipping* bedeutet, dass ein Teil des *Reads* (entweder am Anfang oder am Ende) nicht an das *Referenzgenom* aligniert werden können. Das kann mehrere Gründe haben, z.B. kann an dem Transkript noch der *Poly-A Schwanz* angebracht gewesen sein. Wenn ein Fragment den *Poly-A Schwanz* inkludiert und dieser mit sequenziert wird, so lässt sich der Teil des *Reads* nicht mehr sinnvoll *mappen*. Die Anzahl von *mismatches* und *clipping* sind bereits in dem *Read* vermerkt und werden lediglich für das *ReadPair* aufsummiert.

Ab hier wird angefangen das Ergebnis korrekt zu formatieren.

1. Falls *cgenes* == 0 wird die *gdist* mit in das Ergebnis geschrieben. Wenn es sich um ein *Strang-positives* oder *negatives* handelt wird zusätzlich auf dem *Gegenstrang* nach inkludierenden Genen gesucht und bei einem Ergebnis größer 0 wird der Eintrag mit "antisense:true" vermerkt.
2. Falls *cgenes* > 0 wird *cgenes* und die Regions-Annotation in das Ergebnis übernommen.

Als letztes wird in beiden Fällen noch der *pcr-index* berechnet und mit in das Ergebnis geschrieben. Der *pqr-index* speichert wie oft wir ein *ReadPair* bereits gesehen haben. Wie in 1 beschrieben, werden die Fragmente auf der *Flow Zelle* zuerst durch *PCR* amplifiziert. Dieses Protokoll garantiert nicht eine gleichmäßig starke Amplifikation über allen Fragmenten. Viel mehr kommt es zu starken Diskrepanzen zwischen der Häufigkeit der einzelnen Fragmente. Der *pqr-index* benutzt die *meltedBlocks* der *ReadPairs* als eindeutige Erkennungssignatur und zählt wie oft ein *ReadPair* bereits vorgekommen ist.

Das Ergebnis wird mit einem *BufferedWriter* in die Ergebnisdatei geschrieben und es wird mit dem nächsten *ReadPair* weiter gemacht.

2.2.3 Reduktion der Heap Benutzung

Die *Reads* innerhalb der *BAM* sind nach Startposition sortiert. Dies lässt sich ausnutzen um die Arbeitsspeichernutzung zu reduzieren. Sobald ein neues Chromosom in der *BAM* Datei erreicht ist, wird die *seenEntries* *HashMap* geleert. Zusätzlich wird der zu dem alten Chromosom zugehörige *IntervalTree* mit dessen Genen gelöscht und der *pqr-index* ebenfalls neu initialisiert.

2.3. Korrektheit

3 – Ergebnisse

Für die drei vorgegebenen *BAM* Dateien wurden die *RPKM* Werte berechnet und in Abbildung 2 aufgetragen. Dafür wurde folgende Formel verwendet:

$$RPKM = \frac{R_G}{\frac{G_L}{1000} \cdot S}$$

Dabei ist R_G die Anzahl an *Reads* deren Regions-Annotation auf G verweist, G_L ist die Länge der aufsummierten geschmolzenen Exons des Gens G und S ist der Skalierungsfaktor $\left(\frac{|R_{total}|}{1000000}\right)$.

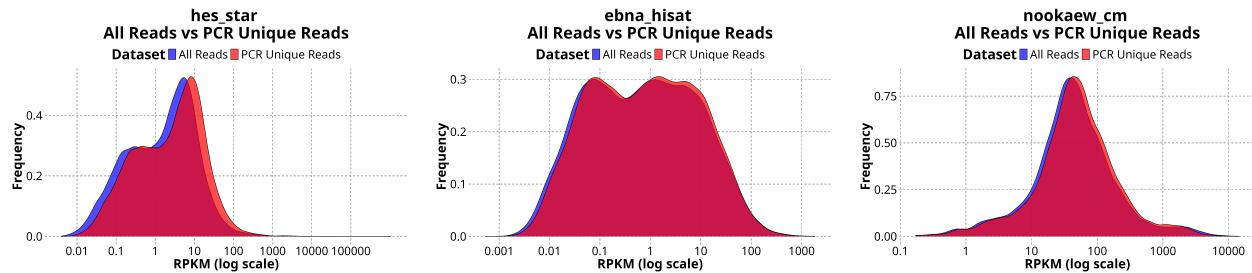


Abbildung 2 – RPKM Verteilung der Gene per Sample. Dabei wird zwischen *All Reads* und *PCR unique Reads* unterschieden. *All Reads* inkludiert alle *ReadPairs* die eine Regions-Annotation besitzen. Bei *PCR unique Reads* werden nur *ReadPairs* mit *pcr-index = 0* berücksichtigt.

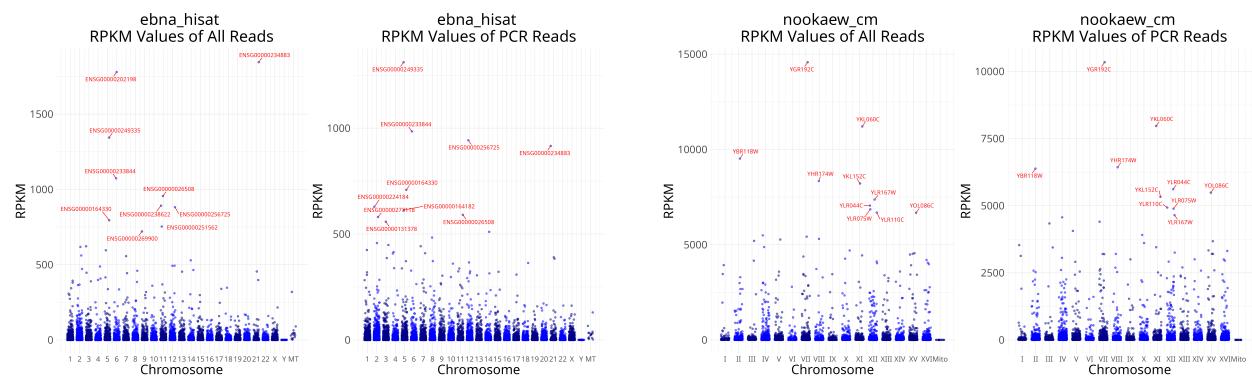


Abbildung 3

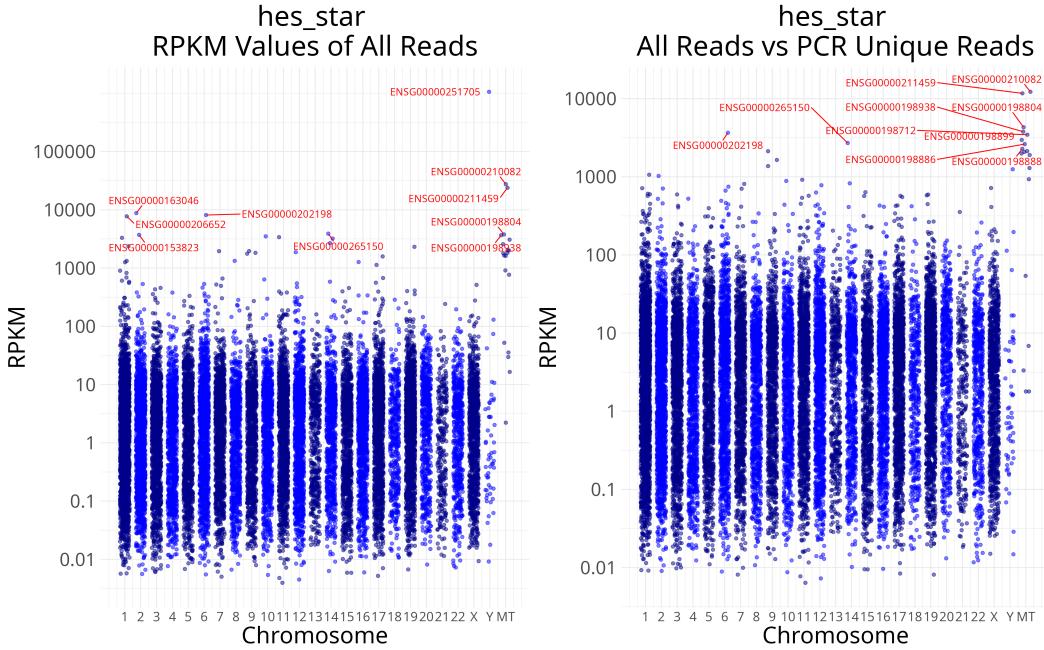


Abbildung 4

3.1. Laufzeit

Die Komplexität der \mathcal{JAR} ist linear in der Anzahl an Einträgen der *BAM* Datei und der *GTF*-Datei, wobei das Einlesen der *GTF-Datei*, bei großen *BAM* Dateien, nur einen Bruchteil G der Gesamtkosten ausmacht. Im worst case sind alle *Reads* in einer *BAM*-Datei mit R vielen Einträgen valide. Somit gibt es insgesamt $\frac{R}{2}$ viele *ReadPairs* welche im worst case alle *genic* sind, d.h eine Regions-Annotation haben. Für jeden dieser *ReadPairs* kostet die gesamte Annotation K viel Aufwand, wobei dieser Aufwand konstant ist. Somit beträgt die Laufzeit im worst case:

$$\mathcal{O}(G + R + \frac{R}{2} \cdot K) \in \mathcal{O}(G + 2 \cdot R \cdot K) \in \mathcal{O}(G + R) \in \mathcal{O}(R).$$

Da $G \ll R$ für große *BAM* Dateien.

3.2. Benchmarking

Die in 3.1 theoretisch bestimmte Laufzeit spiegelt sich in Abbildung 5 wieder. Wenn man zusätzlich die Anzahl an annotierten *ReadPairs* in Tabelle 2 betrachtet, lässt sich grob abschätzen, dass die \mathcal{JAR} pro 20 Mio *ReadPairs* ca. 200s braucht. Sie skaliert sogar etwas besser für noch größere Dateien, denn für > 46 Mio *ReadPairs* in der *ebna_hisat* Datei benötigt sie unter 400s.

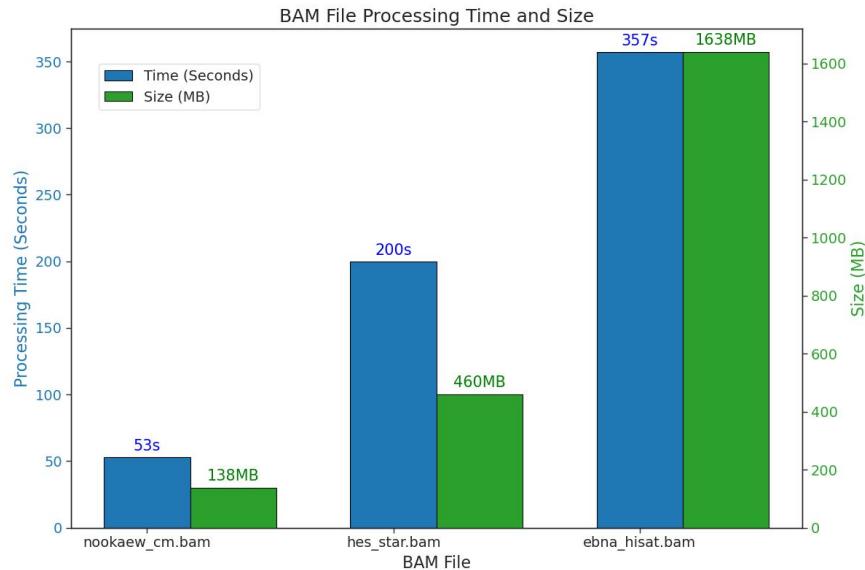


Abbildung 5 – Laufzeit der JAR auf den 3 vorgegebenen *BAMs* in Sekunden verglichen mit dem *BAM* Volumen in *MB*. Die Ausführung der JAR erfolgte auf folgender Hardware (CiP Rechner): Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz mit 6 Kernen (12 Threads), 12 MB L3-Cache, und einer RAM-Limitation von 10 GB (-Xmx10g).

Tabelle 2 – Anzahl an annotierten *ReadPairs* per *BAM* Datei

BAM	Total <i>ReadPairs</i>
ebna_hisat	46536090
hes_star	20098529
nookaew_cm	5054624

Referenzen

BioRender. 2024. *BioRender - Biological Figure Creation Tool*. <https://BioRender.com>. Accessed: 2024-12-09. (Cited on page 6).